Specification and verification challenges for sequential object-oriented programs

Gary T. Leavens, K. Rustan M. Leino, and Peter Müller

TR #06-14b
(KRML 161)
May 2006, revised August 2006, corrected July 2008

Keywords: Program verification, specification, contract, object-oriented programming, challenge, JML, Spec#.
2000 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract, reliability, tools, JML, Spec#; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, pre- and post-conditions, specification techniques;

The original version of this paper appears in *Formal Aspects of Computing*, **19**(2):159-189, June 2007. DOI 10.1007/s00165-007-0026-7.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Specification and verification challenges for sequential object-oriented programs

Gary T. Leavens[1] and K. Rustan M. Leino[2] and Peter Müller[3]
[1]Dept. of Computer Science, Iowa State University, USA. e-mail: leavens@cs.iastate.edu
[2]Microsoft Research, Redmond, WA, USA. e-mail: leino@microsoft.com
[3]ETH Zurich, Switzerland. e-mail: peter.mueller@inf.ethz.ch

July 8, 2008

Abstract

The state of knowledge in how to specify sequential programs in object-oriented languages such as Java and C# and the state of the art in automated verification tools for such programs have made measurable progress in the last several years. This paper describes several remaining challenges and approaches to their solution.

1 Introduction

The last few years have shown a renewed interest in formally verifying software. For example, within the last decade, interactive program verifiers have been applied to control software and other critical applications [2, 26], software model checking has made strides into industrial applications [7], and a number of research tools for bug detection have been built using automatic program-verification technology [23, 47, 49, 74, 96]. In fall 2005, a large, international group of researchers gathered in Zurich at the *Verified Software: Theories, Tools, Experiments* conference [63], to explore the next steps in a long-term initiative by Tony Hoare [62] to advance the science of program construction. Specification and verification of sequential object-oriented software is certainly important to this overall picture.

In this paper, we describe several important specification and verification challenges as we see them today. We draw on our experience with specifying and verifying code using the Java Modeling Language (JML) [77, 78, 82] and Spec# [10, 12, 87], and static checking and verification tools for these [9, 49, 74]. While we do not claim that the set of challenges is complete, we hope that these problems will provide a roadmap for problems to attack and a basis for measuring future progress.

Three years ago, Jacobs, Kiniry, and Warnier described several challenge problems for Java program verification [67]. Here, we group them into the categories used in our paper (plus a category "Others" for the remaining challenges):

A. Data Abstraction in Specification

   1. Specification: numeric models and method calls in specifications

B. Frame Properties

   2. Side effects in expressions

C. Heap Data Structures

   3. Aliasing and field access
   4. Class invariants and callbacks
   5. Static initialization

D. Control Flow

6. Breaking out of a loop
7. Catching exceptions

E. Others

8. Bitwise operations
9. Overloading and dynamic method invocation
10. Inheritance
11. Non-termination

The specification challenge of dealing with computer versus infinite arithmetic (A.1) has been addressed in the context of JML [25]. The problem of how to deal with method calls in specifications (A.1) has also seen some work recently [34, 38, 69], but still faces open issues regarding frame properties. We discuss these in Sec. 3.1. Side effects in expressions (B.2) as well as aliasing (C.3) are handled by various program logics for object-oriented languages [1, 14, 15, 17, 65, 66, 70, 85, 105, 106, 107, 108, 109, 111] and also in ESC/Java [49, 92, 74] and other tools geared to the verification of Java-like programs [9, 23]. There has been much work on invariants and callbacks (C.4) [10, 87, 90, 103], which we discuss in more detail in Sec. 4.1. Although some work has also been done on static initialization (C.5) [88], we still consider this problem an open challenge, which we discuss further in Sec. 3.2. Both challenges related to control flow (D.6 and D.7) are solved by logics for abrupt termination [15, 65] or by translating programs to low-level languages such as BoogiePL [9, 39, 92] before verification. Handling bitwise operations (E.8) is important in practice. To automate verification of bitwise operations, there are various decision procedures [35, 122], but there is still work to be done on integrating these decision procedures with theorem provers, in particular, on combining them with arithmetic operations. Dynamic method binding (E.9) and inheritance (E.10) are largely handled by the discipline of behavioral subtyping [4, 80, 83, 93, 98], which is incorporated into JML and Spec# using the idea of specification inheritance [41, 76, 81, 118]. Finally, non-termination (E.11) can be addressed using standard techniques such as loop variants. The problem has also been tackled successfully using model checkers [36].

In summary the challenges presented by Jacobs, Kiniry, and Warnier's paper are areas where much progress has been made. Still, a number of difficult challenges remain, which are the subject of the present paper.

1.1 Scope and Assumptions

Our survey of these challenges is limited to specification and verification techniques for object-oriented languages, such as Eiffel, Java, and C#. Because we draw on our experience with JML and Spec#, we largely ignore concurrency issues and instead concentrate on issues in the specification and verification of sequential code.

Even within the domain of specification and verification techniques for sequential object-oriented programs, there are several different styles of specification and verification. We focus on detailed design specification, i.e., the specification of interfaces of individual program modules, also known as interface specification [57, 120, 121]. This can be contrasted with requirements specification, which often occurs earlier in the development cycle and is more concerned with the behavior of an entire program and less with the specification of individual modules.

Most interface specification languages use some variation on Hoare's pre- and postcondition technique [5, 54, 60]. A well-known early example of such a language is VDM [71]. The Larch family [57, 120] of interface specification languages exemplifies the approach of writing such pre- and postconditions using a specialized mathematical vocabulary. Specifications operate on abstract values [61], which are abstractions of the "concrete" state of the program. Furthermore, the operations used on abstract values are completely mathematical, and thus an excellent fit for formal manipulation (e.g., with theorem provers).

Unfortunately, experience with Larch-style interface specification languages indicates that a mathematical syntax for assertions, such as the Larch Shared Language [57], which is different than the programming language's syntax, is a barrier to use by programmers. Programmers seem more comfortable with an assertion language that is based on the programming language's own expression syntax. This is the approach followed by Gypsy [3], Anna [94, 95], APP [112], and Eiffel [97, 98], and adopted by JML [79] and Spec# [12].

Several of the challenges we describe in the design of such Eiffel-like interface specification languages stem from this fundamental decision to write assertions using a subset of the expressions in the underlying programming language. One of the basic problems is to overcome the mismatch between the programming language's expressions and the needs of automatic theorem provers.

It is essential that verification techniques are modular, that is, that they allow one to reason about a class independently of its clients and subclasses. Modularity is crucial to verify reusable classes such as library classes and for scalability. Many of our challenges stem from this modularity requirement. They call for modular solutions to problems for which non-modular solutions already exist.

The specification and verification challenges described in this paper are challenges for specification and verification methodology, that is, how to apply existing concepts, formalisms, logics, etc. to specify and verify a program. Therefore, the discussion is somewhat independent of the particular program logic that is used. We believe the discussion applies to all or most of the logics for object-oriented languages, including Hoare-style logics [1, 17, 65, 66, 70, 85, 106, 107, 109], dynamic logics [14, 15], and separation logic [105, 108, 111].

1.2 Outline

In this paper, we describe challenges in the areas of data abstraction in specification (Sec. 2), frame properties (Sec. 3), reasoning about heap structures (Sec. 4), control flow (Sec. 5), and practical considerations (Sec. 6). We describe each challenge and try to give an idea of why it is not yet solved, by describing a number of solution approaches with their potentials and limitations.

2 Data Abstraction in Specification

One of the key innovations in JML's design is the use of a library of *modeling types* that describe mathematical bags (multisets), sets, sequences, relations, and maps [79]. These modeling types play the same role in a JML specification as the built-in traits in the Larch Shared Language or the built-in types of VDM: they allow the specifier to describe abstract values of objects using standard mathematical notions. These types are designed to have immutable objects, to better match mathematics. They allow the specification of abstract values, especially for collection classes. For example, the abstract value of a `java.util.Collection` can be specified using a (specification-only) model field whose type is a `JMLEqualsBag`. While the hope was that such types would be useful for both runtime assertion checking and static verification, they do not work well with static verification. This leads to our first two challenges.

2.1 Specifying Modeling Types

If an interface specification language provides several built-in modeling types, a fundamental problem arises in how to specify them in a way that is useful for verification. Modeling types can, of course, be specified using other modeling types, but ultimately some modeling types must be specified in some other way.

Challenge 1 *Develop a specification technique for modeling types.*

2.1.1 Solution Approach 1: Collected Algebraic Specifications

One approach to solving this challenge is to specify modeling types by mimicing algebraic specification techniques [19, 44, 51, 56, 117]. This approach initially seems sensible, because modeling types typically have immutable objects. Thus, semantic ideas developed for equational algebraic specifications, such as the initial algebra approach [19, 44, 51] or the final algebra approach [117], could be used to give the mathematical meaning to such a specification. Specifiers can use the technique of sufficient completeness [56] to make sure that they have specified a type to the level of completeness desired.

For instance, a data type Set contains laws that relate the operations add and has, such as: $(\forall Set\ s, Element\ e : has(add(s, e), e))$. In a similar way, we can relate the methods of a modeling type. For instance, each instance `s` of a modeling type `ModelSet` has to satisfy the law: `(\forall Object e; s.add(e).has(e))`. In this JML notation, universal quantification starts with `\forall`. (Such expression keywords start with a backslash to prevent them from being confused with programmer-defined names.)

Fig. 1 on the following page shows a specification for `ModelSet` using this approach. In this specification, the invariant states several properties of the methods shown in the figure. Such specifications are used for many of the JML modeling types, and in actual practice they are considerably more extensive than shown in Fig. 1.

Unfortunately, this approach is not usable for verification, due to several problems.

```
public /*@ pure @*/ class ModelSet {

  /*@ public invariant (\forall Object e, e2;
    @       this.add(e).has(e)
    @    && this.add(e).add(e2).equals(this.add(e2).add(e))
    @    && this.add(e).add(e).equals(add(e))
    @    && (this.equals(new ModelSet()) ==> !this.has(e)));
    @*/

  public ModelSet() { /* ... */ }
  public boolean has(Object o) { /* ... */ }
  public ModelSet add(Object o) { /* ... */ }
  public boolean equals(/*@ nullable @*/ Object o) { /* ... */ }
}
```

Figure 1: A JML modeling type `ModelSet` that is specified using an algebraic specification technique. In JML, annotations are enclosed in special comments that start with an at-sign (`@`); in such annotations initial at-signs on a line are ignored. They keyword **`pure`** says that the methods of this type have no side effects. An **`invariant`** declares a property that is true in all visible states. The operator **`==>`** means implication. The modifier **`nullable`** allows the parameter to `equals` to be null, contrary to JML's default.

A first problem is that the invariant in Fig. 1 quantifies over all objects, making it non-executable and thus unsuitable for runtime assertion checking. With static verification techniques, this invariant's quantification would mean that every new object created could potentially violate the invariant. It is not clear how to reason efficiently about such invariants. Such additional proof obligations would be non-modular and impractical. In our example, creating a new object in fact cannot violate the invariant. Therefore, we would like to tell the theorem prover not to check every new object against this invariant. However, taking such invariants as axioms instead of as invariants seems dangerous, as it may lead to inconsistencies and would not be checked against the implementation.

A second problem is that the approach shown in Fig. 1 uses the `equals` method of the `ModelSet` type, whereas standard algebraic specification techniques work with a built-in notion of equality. The use of the `equals` method introduces several considerations that are not problems in standard algebraic data type specifications, such as that equality might not be a congruence (reflexive, symmetric, transitive, preserved by the methods). Furthermore, calls to the `equals` method make theorem proving more difficult (see Sec. 3.1).

2.1.2 Solution Approach 2: Specifying Methods via other Methods

It seems possible to solve the problems of the first approach while still using ideas inspired by algebraic specification techniques, in that methods are specified in terms of their effect on other methods. However, this second approach does not use a single algebraic style specification in an invariant. Instead, it divides the set of methods up into query and non-query methods; no specifications are written directly for query methods, but each non-query method's behavior is specified by describing how it changes the results of all query methods [98]. (One can also distinguish between primitive and derived query methods, to help abbreviate such specifications [56].)

As an example, consider the type `UModelSet` in Fig. 2 on the following page. In this example, the meaning of the non-query methods `emptySet` and `add` are both specified using the query method `has`. The method `has` itself is not directly given a specification.

The main drawback of this approach seems to be that the verification of an implementation of a modeling class may need additional specifications. For example, to verify an implementation of the `has` method in `UModelSet`, one would have to write a (private) implementation-dependent specification.

This approach also requires a sound treatment of quantifiers, such as the one in the specification of `emptySet` (see Sec. 2.2), and of method calls in specifications (see Sec. 3.1).

```
public /*@ pure @*/ interface UModelSet {

  public boolean has(Object o);

  /*@ ensures \result.has(o)
    @       && (\forall Object e; e != o ==> this.has(e) == \result.has(e));
    @*/
  public UModelSet add(Object o);

  //@ ensures (\forall Object e; !\result.has(e));
  public UModelSet emptySet();
}
```

Figure 2: A JML modeling type `UModelSet` that is specified by giving specifications of various methods in terms of other methods. The keyword **`ensures`** starts a postcondition for the following method. The keyword **`\result`** stands for the result of a (non-void) method.

2.1.3 Solution Approach 3: Translation between Modeling Types and Mathematical Theories

Another approach for specifying modeling types in some technique other than that used to specify normal, user-defined types. One possibility is to develop an automatic translation between modeling types and theories of some standard theorem prover (as suggested in [79]). Such a translation could be defined either from modeling types to mathematical theories or vice versa.

Translating modeling types into mathematical theories allows programmers to write new modeling types and then automatically translate them for static verification. This seems possible, but is non-trivial [99]. First, the resulting data types are typically complicated. For instance, JML's model class `JMLObjectSet` for sets of references cannot simply be mapped to a mathematical set of references. `JMLObjectSet` contains several ghost fields, which are independent coordinates in the state space. Therefore, the resulting data type is a tuple, where one component is a mathematical set. This makes working with it tedious. Second, if the modeling type is not final, then programmers can override the `equals` method in various ways. Therefore, the `equals` method of a non-final modeling type cannot always be translated to mathematical equality. The same problem occurs when programmers implement their own modeling types and `equals` methods. Third, modeling types may refer to methods of normal program code. For instance, the `has` method of `JMLValueSet` uses the `equals` method to determine whether the (non-model) argument is in the set. It is not clear how such calls to program code can be expressed in a mathematical theory (without formalizing the whole language semantics as part of the theory).

Translating mathematical theories into modeling types also has shortcomings. First, if programmers want to develop their own modeling types, they would have to specify them in the language of a theorem prover. This is contrary to the basic idea of contract languages, which try to shield programmers from specialized theorem prover notation. Second, runtime assertion checking requires that modeling types be executable. However, arbitrary mathematical theories cannot be translated automatically into executable program code.

In summary, it seems that the generality of these translation approaches causes problems. In particular, it is unclear how to verify that the implementations (used by a runtime assertion checker) of such modeling types are correct with respect to the mathematical theories (used by a theorem prover or other static analysis tool).

2.1.4 Solution Approach 4: Built-in Modeling Types Based on Mathematical Theories

The problems of the automatic translations between modeling types and mathematical theories can be avoided if the specification language provides a fixed, but carefully chosen set of modeling types together with their translations to mathematical theories. Programmers could then develop new modeling types only by instantiating existing ones. In essence, this approach would harken back to VDM [71], in using a small set of built-in types for specification.

This approach meshes with the very recent work of Charles [27]. In Charles's work on JML, the Java "native" keyword is reused to declare methods and types that have a correspondence in the prover (Coq). Types declared

as native are considered to be value types, that is, not to be subtypes of `Object`. Also, the prover has a built-in understanding of how certain Java types (such as `Object`) correspond to types in the prover (e.g., `Reference`). That is, when translating from JML to the prover's language, all occurrences of such Java type are replaced by their corresponding prover type. Methods of such native types must also be "native" and are treated as uninterpreted function symbols in the theorem prover. The JML declaration of such native types and methods is accompanied by text in the prover's native language that axiomatizes the corresponding uninterpreted function symbols. Normal users would not extend the set of such types, and no extra ghost or model fields would be permitted in such types.

This seems like a promising approach to solving the challenge. One problem is defining a set of such "native" types that would satisfy the demands of a wide variety of theorem provers and that would also work for runtime assertion checking. In particular, correctness of the implementations of "native" types with respect to the mathematical theories is again an issue. Another problem is figuring out how to use "native" types to specify types such as Java's collection types, where program code (`equals`) is supposed to define membership in the collection.

2.2 Quantifiers and Comprehensions

It is well known that various generalized quantifiers (such as summations and products) are useful in specifications [33, 55]. Similarly, mathematicians have long found that set comprehensions are very convenient notational abbreviations. Haskell [64] and other functional languages also use comprehension notations for lists to great effect.

Generalized quantifiers and comprehensions are equally useful and important in specification languages, where generalized quantifiers are notational shorthands and comprehensions act as literals for modeling types. A prominent example of the utility of comprehension notations is Z [115], in which, for example, set comprehensions are a central and important feature. JML has a few kinds of generalized quantifiers and set comprehensions. The example in Fig. 3 on the next page shows the use of a generalized quantifier `\num_of`, which counts the number of integers, `j`, that both satisfy the range predicate, `0 <= j && j < args.length`, and the body predicate `args[j].startsWith("-")`. This numerical quantifier is used in a loop invariant. The loop invariant is needed to show that `count` is within the boundaries of the `out` array, which illustrates that quantifiers can be useful even in situations where one's verification ambitions are limited, like trying only to prove the absence of array index bounds errors; of course, quantifiers and comprehensions are even more useful if one is trying to prove full functional correctness.

Quantifiers and comprehension expressions pose several pitfalls for specification language designers. For example, if quantifiers only quantify over non-garbage objects, then they become sensitive to garbage collection, which causes semantic problems [24]. On the other hand, quantifying over non-allocated objects is also problematic. For instance, if the quantifier in the invariant of class `ModelSet` (Fig. 1 on page 4) ranges over all objects including non-allocated objects, then the invariant calls methods with parameter objects that are not allocated. It is unclear what it means if these methods access the state of these parameter objects.

There are also practical difficulties in the implementation of quantifiers for runtime assertion checking, which require either restriction of the language [73, 75] or recognition of patterns of bounded quantification [28, 116]. Similar difficulties affect comprehension expressions. However, these difficulties in language design and runtime assertion checking seem fairly well understood.

The remaining challenge is about program verification.

Challenge 2 *Develop a verification technique for general quantifiers and comprehensions that is suitable for automatic verification systems.*

This challenge focuses on automatic program verifiers, such as ESC/Java and Boogie. These encode the proof obligations as first-order formulas that are passed to an automatic theorem prover like Simplify [40]. In such automatic first-order provers, common inductive definitions of generalized quantifiers are not readily available.

2.2.1 Solution Approach: Replace Comprehensions by Functions

In our example, we could introduce the side-effect free (*pure*) method shown in Fig. 4 on the next page to count the number of elements in the array `a` from index `from` that start with `"-"`. We use this method to replace the comprehension in the loop invariant of method `filter` as follows:

```
//@ loop_invariant 0 <= count && countHits(args,i) == out.length - count;
```

```
public class Comprehension {
  private static String[] filter(String[] args) {
    int count = 0;
    for(int i = 0; i < args.length; i++) {
      if(args[i].startsWith("-")) { count++; }
    }
    String[] out = new String[count];
    count = 0;

    /*@ loop_invariant 0 <= count
      @   && (\num_of int j; 0 <= j && j < args.length;
      @                          args[j].startsWith("-"))
      @      == out.length - count;
      @*/
    for(int i = 0; i < args.length; i++) {
      if(args[i].startsWith("-")) {
        out[count] = args[i];
        count++;
      }
    }
    return out;
  }
}
```

Figure 3: A JML example that uses a generalized quantifier **\num_of** in its loop invariant. Details of this expression are explained in the text.

```
  /*@ requires 0 <= from;
    @ ensures  a.length <= from ==> \result == 0;
    @ ensures  from < a.length  ==>
    @            \result == (a[from].startsWith("-") ? 1 : 0) + countHits(a, from+1);
    @*/
  /*@ pure @*/ static int countHits(String[] a, int from) {
    int n = 0;
    for(int i = from; i < a.length; i++) {
      if(a[i].startsWith("-")) { n++; }
    }
    return n;
  }
```

Figure 4: A method, `countHits`, that could be used to avoid the **\num_of** quantifier in the previous figure. The **requires** clause specifies its precondition. The use of two **ensures** clauses is equivalent to the conjunction of the postconditions they specify.

However, this solution has some shortcomings. First, it requires a technique for reasoning about method calls in specifications (see Sec. 3.1). Second, specifiers generally have to introduce auxiliary methods with non-trivial (typically recursive) specifications for each quantifier or comprehension, which increases the specification overhead significantly. One possible line of attack might be to develop heuristics that cause the verification-condition generation to introduce suitably axiomatized functions whose parameters are the variables mentioned in the generalized quantifier.

Without a solution to Challenge 2, users would have to choose between automatic theorem proving support and specifications that are rich enough to mention generalized quantifiers and comprehensions.

3 Frame Properties

This section presents several challenges related to frame properties. Frame properties say what a method is permitted to change during its execution [18]. The permitted modifications are often specified in so-called "modifies clauses" [57]. Our JML examples use "assignable clauses" to say what locations a method may assign. For instance, the assignable clause of method `push` in Fig. 5 on the following page permits the method to assign to all fields of its receiver, but nothing else. Assignable clauses are slightly more restrictive than modifies clauses. A method that assigns to a location l and then re-establishes its original value still has to list l in its assignable clause, but since the method in effect does not modify l, l need not be listed in the modifies clause. The challenges presented in this section do not rely on this subtle difference.

3.1 Method Calls in Specifications

Assertions in Eiffel, JML, and Spec# rely on pure (that is, side-effect free) methods, so-called *observers*, to support data abstraction. For instance, the `BoundedStack` interface in Fig. 5 on the next page contains an observer method `hasRoomFor`, where `stack.hasRoomFor(i)` yields true if and only if `stack` has room for at least `i` additional elements. This observer is used to provide a specification for method `push` without referring to the concrete implementation of the stack. Implementation independence is required by information hiding. Moreover, implementation independence is crucial to supporting subtyping since different subtypes must satisfy a common specification, but may have different implementations (or no implementations at all in the case of abstract classes and interfaces).

Recent papers by Cok [34] as well as by Darvas and Müller [38] present encodings of observer methods in program logics. However, they do not explain how to reason about frame properties when the specification uses observer methods.

Existing specification techniques for frame properties [78, 89, 90, 102] allow one to describe the fields that are potentially modified by a method execution using an assignable clause. However, assignable clauses do not specify the effects of a method execution on the results of observers. In our example, method `push` affects the result of `hasRoomFor` for some arguments, but this effect is not declared in `push`'s assignable clause.

Since effects on observers are not covered by assignable clauses, the specification of method `getOperand` of class `Calculator` does not express whether the result of `stack.hasRoomFor` is potentially affected by the method. In fact, the specification in Fig. 5 on the following page does not prevent such an interaction between `getOperand` and `stack.hasRoomFor`. Class `StrangeStack` in Fig. 6 on the next page stores the stack elements in the unused part of the operand array of the `Calculator` object `calc`. Consequently, modifications of `calc.next` affect the capacity of the stack and, therefore, the result of `hasRoomFor`. If a `Calculator` object c and a `StrangeStack` object mutually use each other, then calling c`.getOperand` may indeed affect the result of c`.stack.hasRoomFor`.

Due to its requires clause, method `constOp` may assume `stack.hasRoomFor(1)`. However, we cannot conclude that this property still holds when it is needed to satisfy the requires clause of the call to `stack.push`, because that property might have been invalidated by the preceding call to `getOperand`. Consequently, we cannot prove that the requires clause of `push` is satisfied, which causes the verification of `constOp` to fail. This example illustrates an open challenge.

Challenge 3 *Develop a specification and verification technique that allows one to determine the effects of heap modifications on the results of observer methods.*

```
public interface BoundedStack {
  /*@ pure @*/ boolean hasRoomFor(int i);

  //@ requires   hasRoomFor(1);
  //@ assignable this.*;
  void push(int i);

  /* other methods omitted */
}
```

```
public class Calculator {
  /*@ spec_public @*/ BoundedStack stack;
  int[] operands;
  int next;

  //@ requires stack.hasRoomFor(1);
  public void constOp() {
    int op = getOperand();
    stack.push(op);
  }

  //@ assignable next;
  int getOperand() {
    int res = operands[next];
    next++;
    return res;
  }

  /* other class members omitted */
}
```

Figure 5: Interface `BoundedStack` uses the pure method `hasRoomFor` to provide an implementation-independent JML specification for `push`. The notation **`this.`**`*` in `push`'s assignable clause means that all fields of **`this`** (including the inherited model field `objectState` and its data group) are assignable. Class `Calculator` uses a `BoundedStack` to store a stack of operands. The method `constOp` fetches an operand and pushes it onto the stack. The modifier **`spec_public`** allows the field `stack` to be used in public specifications.

```
public class StrangeStack implements BoundedStack {
  Calculator calc;
  int count;

  /*@ pure @*/ public boolean hasRoomFor(int i) {
    return i <= calc.next - count;
  }
}
```

Figure 6: A possible implementation of interface `BoundedStack`. The stack elements are stored in the unused part of the operand array of the `Calculator` object `calc`. Consequently, modifications of `calc.next` affect the capacity of the stack and, therefore, the result of `hasRoomFor`.

3.1.1 Solution Approach 1: Listing Modified Observers

One approach to this challenge is to require a method's assignable clause to list all observers that are potentially affected by the method. This can be done in COLD-K [46, section 5.7], where the frame of a procedure specification lists the variables, including the COLD-K equivalent of observer methods, whose value may be changed by the procedure.

However, this solution is obviously not modular.[1] To see why, consider a class `Sequence` with an observer method `isEqualTo(BoundedStack b)`, which states whether the sequence and the stack contain the same values. Method `push` affects the result of `isEqualTo` by adding an element to the stack, but since class `Sequence` might have been developed long after `BoundedStack`, method `push` cannot be required to declare its effect on `isEqualTo`.

Besides not being modular, listing modified observers is also too weak since it handles only heap changes through method calls. However, field updates also change the heap and, therefore, potentially affect the result of an observer. Consider a variant of the `Calculator` example, where the implementation of `getOperand` is inlined into the body of the method `constOp`. In this case, it is again not possible to prove that `constOp`'s call to `push` satisfies the requires clause of `push` because we cannot prove that the assignment to the field `next` that now precedes the call to `push` does not affect the result of `hasRoomFor`.

3.1.2 Solution Approach 2: Model Fields

Model fields [30, 84] are specification-only fields whose value is determined by applying a mapping (a representation function) to the concrete state of an object. Therefore, model fields are similar to observers, but are restricted in two ways. First, model fields do not have parameters. Second, since a model field encodes an abstraction of an object, it is reasonable to require model fields to be confined [89, 102]. The value of a *confined* model field may depend only on the state of the receiver object including the sub-objects of an aggregate object. We assume that the sub-object relation is acyclic and is declared explicitly in programs, for instance, by using ownership annotations [32, 87]. Observers typically serve a more general purpose than model fields. Therefore, the result of an observer method may depend on the state of all reachable objects. For instance, an `equals` observer may depend on the state of its receiver and on the state of its explicit parameter.

The confinededness of model fields allows one to specify frame properties for model fields in a modular way [89, 102]. The modification of an object x potentially affects: (1) model fields of x and (2) model fields of aggregate objects containing x as sub-object. Model fields of group 1 can be listed in assignable clauses. With appropriate alias control, model fields of group 2 cannot be accessed by methods of x and, therefore, do not have to be listed in assignable clauses [89, 102]. Note that the modularity problem of the `isEqualTo` example above stems from the fact that this observer depends on the state of its explicit argument and is, therefore, not confined.

The similarity between model fields and observers suggests that the existing verification techniques for model fields can be generalized to confined, parameterless observers. Fig. 7 on the following page shows a variant of the `BoundedStack` example, where the observer `hasRoomFor` has been replaced by two confined, parameterless observers `getSize` and `getCapacity`. In this example, we treat confined, parameterless observers like model fields, that is, we require them to be listed in the assignable clause of each method that potentially affects their result values.

Since `getSize` and `getCapacity` are confined, their results depend only on the state of the receiver and its sub-objects. The **`rep`** annotation in the declarations of the fields `stack` and `operands` expresses that the stack and operand array are sub-objects of the `Calculator` object. Since the sub-object relation is acyclic, we know that a `Calculator` object c is not a sub-object of the `BoundedStack` object $c.\texttt{stack}$. Consequently, we can prove that an execution of $c.\texttt{getOperand}$ does not affect the values of $c.\texttt{stack.getSize}$ and $c.\texttt{stack.getCapacity}$. Hence, they need not be listed in `getOperand`'s assignable clause, and so we can conclude that the call, in `constOp`, to `getOperand` does not invalidate the requires clause of `push`.

Soundness and modularity of existing verification techniques for model fields rely on the confinededness of the model fields, because this allows one to determine whether a heap modification might affect the value of a model field. Since heap modification does not affect parameter values, we expect that one can generalize these techniques to confined observers with parameters such as `hasRoomFor`. This generalization supports, for instance, an `equals` observer that tests for reference equality. Such an observer uses the parameter value, but does not read the state of the explicit parameter. It is, therefore, confined. However, it does not support a version of `equals` that tests for deep

[1] COLD-K also uses the third approach, described below, to deal with this modularity problem.

```
public interface BoundedStack {
  /*@ pure confined @*/ int getSize();

  /*@ pure confined @*/ int getCapacity();

  //@ requires
getSize() < getCapacity();
  //@ assignable this.*, getSize;
  void push(int i);

  /* other methods omitted */
}
```

```
public class Calculator {
  /*@ spec_public rep @*/
                BoundedStack stack;
  /*@ rep @*/ int[] operands;
  int next;

  /*@ requires stack.getSize() <
    @          stack.getCapacity();
    @*/
  public void constOp() {
    int op = getOperand();
    stack.push(op);
  }

  //@ assignable next;
  int getOperand() {
    int res = operands[next];
    next++;
    return res;
  }

  /* other class members omitted */
}
```

Figure 7: Alternative JML specification of interface `BoundedStack`. The observers `getSize` and `getCapacity` yield the number of elements and the capacity of the stack, respectively. They are parameterless and confined and can, therefore, be treated like model fields. The **`confined`** modifier is supported by Spec#, but not by the current version of JML. Listing observers in assignable clauses such as in the specification of `push` is currently not allowed in either Spec# or JML.

equality, which requires read access to the state of the explicit parameter. Therefore, this approach is promising for many practical applications, but not a complete solution to the challenge.

3.1.3 Solution Approach 3: Read Effects

The solution approach based on model fields requires that observers read only the state of the receiver object and its sub-objects. A verification technique can use this information about the read effect of an observer to determine which heap changes (or write effects) potentially have an impact on the result of an observer. The *read effect* of a method m is the set of all mutable heap locations that are potentially read by m [53]. Analogously, the *write effect* of m is the set of all heap locations that are potentially modified by m.

In general, one can prove that a method m does not affect the result of an observer o if the write effect of m and the read effect of o are disjoint. While specifications typically describe the write effect of a method in an assignable clause, read effects are usually not specified explicitly.[2] In the following, we discuss three approaches to using read effects for reasoning about observers.

Mutable State Independent Observers. We call an observer *mutable state independent* if its result does not depend on any mutable state. In other words, the read effect of a mutable state independent observer is the empty set. Therefore, the result of a mutable state independent observer cannot be affected by any heap changes, which simplifies reasoning significantly.

Consider class `ArrayStack` in Fig. 8 on the next page, which implements the `BoundedStack` interface from Fig. 7 on the preceding page. The observer `getCapacity` is declared (mutable) state independent because it reads only immutable fields, namely `elems`, which is immutable because it is final[3], and `elems.length`, which is immutable because the size of arrays cannot be changed in Java.

Since `getCapacity` is mutable state independent, its result cannot be affected by heap changes, in particular, by the execution of `getOperands` in the `Calculator` example (Fig. 7 on the previous page).

Mutable state independent methods solve Challenge 3, but only for observers that do not read any mutable state, such as many mathematical operations, and observers that operate on immutable state, such as most methods of Java's `String` class. However, they do not solve the challenge in general. For instance, method `getSize` of class `ArrayStack` is not mutable state independent. Therefore, this approach does not suffice to verify method `constOp` of class `Calculator`.

Complete Specifications of Result Values. The relevant read effect of an observer can be determined if the observer has a complete specification of its result value, that is, an ensures clause of the form `\result == E`, where the expression E refers to parameters, fields, and other observers with complete specifications of their result values. For instance, method `getSize` in Fig. 8 on the following page completely specifies the result in terms of the field `count`. With a complete specification of the result value, a conventional assignable clause is sufficient to determine whether a method affects the result of an observer. In our `Calculator` example (Fig. 7 on the previous page), the assignable clause of `getOperand` does not mention `stack.count`. Therefore, `getOperand` must leave `stack.count` unchanged. From this information and the ensures clause of `getSize` in class `ArrayStack`, we can conclude that the result of this observer is not affected. If class `Calculator` were to use class `ArrayStack` instead of the interface `BoundedStack`, then one could use the stronger specification of `getSize` and `getOperand` to verify method `constOp`.

The drawback of complete specifications of result values is that they are difficult to write in an implementation-independent way. For instance, in class `ArrayStack`, the ensures clause of `getSize` violates information hiding by mentioning the private field `count` (hence, in JML, `count` must be declared to be **`spec_public`**). This ensures clause would have to be expressed using a model field [30] in the interface `BoundedStack` (Fig. 7 on the preceding page), since the field `count` cannot be declared in the interface. Furthermore, different subclasses of `BoundedStack` might implement and specify `getSize` in different ways. Thus, complete specifications of result values are only a partial solution to Challenge 3 and, in general, require additional support for data abstraction such as model fields.

[2] However, JML does have an "accessible clause" that allows specification of read effects [82, Section 9.9.10].

[3] We assume that an observer is called only on fully initialized receiver objects. Therefore, it is safe to consider final fields as immutable state because their value cannot be mutated after the object is initialized.

```
public class ArrayStack implements BoundedStack {
  final int[] elems;
  /*@ spec_public @*/ int count;

  //@ also
  //@   ensures \result == count;
  public /*@ pure @*/ int getSize() {
    return count;
  }

  public /*@ pure state_independent @*/ int getCapacity() {
    return elems.length;
  }

  //@ also
  //@   requires   getSize() < getCapacity();
  //@   assignable this.*;
  public void push(int i) { /* ... */ }

  /* other methods omitted */
}
```

Figure 8: An implementation of interface `BoundedStack`. The observer `getCapacity` is mutable state independent since it reads only immutable fields. The **`state_independent`** modifier is supported by Spec#, but not by the current version of JML. In JML the keyword **`also`** must be used to start a specification for an overriding method; it says that the given specification is joined with that of the method being overridden.

Explicit Effect Specifications. The approaches discussed so far either work with a very coarse specification of read effects (confined and mutable state independent observers) or infer read effects from ensures clauses. These approaches are useful for special cases, but do not solve Challenge 3 in general. A more comprehensive solution can be achieved by specifying the read effects of methods explicitly (see [46, section 10.11] [53, 69] for effect specifications).

Explicit specifications of read effects must be implementation independent to support interfaces and information hiding. Clarke and Drossopoulou achieve that by building an effect system on top of an ownership type system [31]. Ownership type systems organize the heap hierarchically. This hierarchy of objects can be used in effect specifications. For instance, the JML-style effect specification "`accessible \under(this)`" could express that the method may read the state of its receiver and its sub-objects, that is, that the method is confined. In interface `BoundedStack` (Fig. 7), we could annotate `getSize` and `getCapacity` with this read effect. Since the `Calculator` object is not a sub-object of the `BoundedStack` object (but vice versa), this read effect allows us to prove that the implementation of `getOperand` does not affect the results of these observers.

Data groups [86, 91] enable more fine-grained effect specifications. A data group is essentially a named collection of heap locations. Listing a data group in a read effect specification allows the method to read all fields in the data group without exposing the names of these fields. Subtyping is supported by allowing subtypes to add new fields to inherited data groups. In our example, `getSize` could be declared to read the `size` data group. The implementing class, `ArrayStack`, would then declare `count` to be a member of data group `size`, which gives `ArrayStack`'s implementation of `getSize` the right to read `count`. A linked list implementation of `BoundedStack` would put all list nodes into the data group to allow `getSize` to iterate over the list and count the number of elements.

As originally described by Leino [86] (and as currently found in JML), a location may be declared to be a member of more than one data group. Greenhouse and Boyland [53] observe that while this is fine for limiting what a method may write (using assignable clauses), it does not give one a sound way to decide on possible interference between methods, which is needed to solve this challenge. For example, in Fig. 5 on page 9, knowing the read effects of `hasRoomFor` and the write effects of `getOperand` does not let one soundly say that `getOperand` does not interfere with `hasRoomFor` if one is permitted to add new locations that are in both sets of data groups.

Recent work by Jacobs and Piessens [69] shows that explicit effect specifications seem to be the most promising approach to Challenge 3. Specifying read effects might seem cumbersome, but the verification of multi-threaded programs also requires these specifications [52, 110], which may justify the additional effort.

3.2 Modification of Static Fields

Most of the state of an object-oriented program resides in the fields of objects, but there are also situations where some state is shared among all instances of a class. In those situations, one can use global variables, or *static fields* as Java and C# call them. A problem arises in reasoning about when static fields change.

The program in Fig. 9 on the following page declares an abstract class whose overridable `run` method performs an operation of some sort. The public method `perform` invokes `run`, bracketing the invocation with calls to `now`, which retrieves the current time. The class keeps track of the number of operations that have completed since the counters of the class were last reset, and also keeps track of the sum of the times elapsed during those operations. Method `perform` ends by computing and printing the average time elapsed during an operation.

The correctness of the implementation of `perform` depends on `operations` being non-zero at the time the average is computed, thus avoiding a division-by-zero error. The class invariant states that `operations` is at least zero, so the increment will make `operations` positive. The implementation of `perform` is therefore correct, provided the second call to `now` has no effect on the static field `operations`, more specifically that it does not set it to zero. Such an undesired effect could, for instance, occur if an override of `now` in a subclass of `Operation` would call `reset`. This shows that method specifications must express which static fields are potentially modified by the method.

The effect of a method on instance fields is described by the method's assignable clause. However, stipulating that a method also affects only those static fields listed in the method's assignable clause has a couple of fatal flaws. First, the discipline violates information hiding, because public methods would have to advertise any private static fields that they modify. Second, the assignable clause of a method would become overly verbose, because it would have to include all static fields modified by all transitive callees. The need to declare modifications of both private static fields and static fields modified by transitive callees in assignable clauses is due to potential reentrancy. Consider for instance a method `C.caller` that calls `D.middle`, which in turn calls `C.callee`. If `C.callee` modifies a private static field of class `C`, then this modification has to be advertised to `C.caller` and, therefore, must be declared

```
import java.util.Date;
public abstract class Operation {
  //@ public model JMLDataGroup runGroup;

  private /*@ spec_public @*/ static long operations;
  private /*@ spec_public @*/ static long elapsedTime;
  private static Date date = new Date();

  //@ public static invariant 0 <= operations;

  //@ assignable runGroup;
  protected abstract void run();

  //@ assignable operations, elapsedTime, runGroup;
  public void perform() {
    long start = now();
    run();
    operations++;
    elapsedTime += now() - start;

    long avg = elapsedTime / operations;
    System.out.println(avg);
  }

  //@ assignable operations, elapsedTime;
  //@ ensures    operations == 0 && elapsedTime == 0;
  public static void reset() {
    operations = 0;
    elapsedTime = 0;
  }

  protected long now() {
    return date.getTime();
  }
}
```

Figure 9: A Java class with some static fields. In this example, the correctness of method `perform` relies on the fact that executions of `now` do not change the value of `operations`. The field `runGroup` is a **`model`** field, and hence only usable in specifications. The type `JMLDataGroup` is used in such declarations to declare data groups.

in the assignable clauses of both `C.callee` and `D.middle`. Dealing with the information hiding problem and the transitivity problem is an open challenge.

Challenge 4 *Develop a specification and verification technique that allows one to determine the effects of methods on static fields.*

For instance fields, the information hiding problem and the transitivity problem are addressed by data groups [86] and ownership [32]. Our solution approaches are to adapt these concepts to static fields.

3.2.1 Solution Approach 1: Data Groups

As explained in Sec. 3.1.3, data groups allow one to group several heap locations into one named collection. Data groups support information hiding in assignable clauses by the following rule. The license to modify a data group implies the license to modify the variables it contains. For example, in Fig. 9, the fields modified by `run` are specified by the data group `runGroup`. Data groups also allow subclasses of `Operation` to introduce more instance fields and declare that these are contained in `runGroup`. Thus, the assignable clause of `run` is both expressive and concise.

One can also attempt to use data groups to solve the information hiding problem for static fields. For example, class `Operation` could declare a data group

```
//@ public static model JMLDataGroup staticGroup;
```

By declaring `operations` and `elapsedTime` to be contained in `staticGroup`, as in:

```
private static long operations; //@ in staticGroup;
private static long elapsedTime; //@ in staticGroup;
```

they no longer need to be declared as **`spec_public`** and the assignable clause of `perform` can be replaced by

```
//@ assignable staticGroup, runGroup;
```

which does not mention any private fields.

To address the transitivity problem, we must also declare nested containments of data groups. Suppose class `Date` declares a data group `Date.staticGroup` and lists it in the assignable clause of method `getTime` (for instance, because the method updates a cache stored in a static field). Then, in order for `perform` to own up to modifying `Date.staticGroup`, we must arrange for that data group to be contained in `Operation.staticGroup`.

Nesting of data groups is also necessary to handle dynamic method binding. Consider for instance a dynamically-bound method `m` declared in a class `C`, and a subclass `D` of `C` that overrides `m`. Let's assume that `C.m` and `D.m` assign to static fields of `C` and `D`, respectively. `C.m` lists a data group `C.staticGroup` in its assignable clause, but not `D`'s data groups because in a modular setting, `C` cannot be expected to know all of its subclasses. Behavioral subtyping requires that `D.m` satisfy the assignable clause of `C.m`. Therefore, the only way for `D.m` to be allowed to modify the static fields in `D` is by declaring `D.staticGroup` to be contained in `C.staticGroup`.

To be useful, it must be possible, at verification time, to know that certain data group containments are *not* present in a program. For example, in order to determine that the call to `Date.getTime` in Fig. 9 on the previous page does not have an effect on `operations`, one needs to determine that `Date.staticGroup` does not (directly or transitively) contain `operations`. Assume for instance that our specification technique would allow a class `Illegal` to declare a data group to contain `Operation.staticGroup` and to be contained in `Date.staticGroup`. With this declaration, `operations` would be transitively contained in `Date.staticGroup`, but this containment could not be determined in a modular way from the declarations of `Operation` and `Date`.

This example shows that a specification discipline needs to restrict where data group containments can be declared: it can either make declarations of the form "data group `g` contains `x`" possible as part of the declaration of `g` (discipline 1) or make them possible as part of the declaration of `x` (discipline 2). With discipline 1, one can determine modularly all static fields contained in `g` by following the containment relation starting from `g`. With discipline 2, one can determine modularly all data groups that contain `x` by following the inverse containment relation starting from `x`. However, neither of the two declarations in isolation is expressive enough to handle our examples: discipline 1 requires class `C` to declare that `C.staticGroup` contains `D.staticGroup`, but `C` cannot be expected to know all of its subclasses; discipline 2 requires class `Date` to declare that `Operation.staticGroup` contains `Date.staticGroup`, but `Date` cannot be expected to know all of its clients. Permitting a mix of both disciplines leads to the modularity problem illustrated by class `Illegal`.

In conclusion, using data groups to specify the modification of static fields cannot be made to fit the requirements on data groups imposed by sound modular verification.

3.2.2 Solution Approach 2: Class Ordering

For instance fields, the transitivity problem of assignable clauses can be solved using ownership. For example, the Boogie methodology [10, 87] allows a method to modify fields of objects directly or transitively owned by the receiver without declaring these fields in the assignable clause. The intuition behind this rule is that owned objects are sub-objects that belong exclusively to their owner. Therefore, client code does not need to know about their modification.

Ownership cannot directly be used for static fields because it prescribes a tree order of exclusive ownership. In contrast, classes are typically global, that is, not exclusively owned by other classes. There is a variation of the Boogie methodology that removes the restriction that entities be ordered by a tree according to the ownership relation. By imposing some further restrictions on what variables may be named in invariants, this variation is able to allow any partial ordering between entities. The variation has been developed in the context of invariants over static fields [88] where the entities are classes.

We assume that the edges in the partial order among classes are declared explicitly. Typically, a class C succeeds class D in the order if C is a client of D or if D is a subclass of C. In the former case, the edge is declared in the client C; in the latter case, it is declared in the subclass D. This shows that the methodology permits a mix of the two disciplines described in Sec. 3.2.1. Soundness is restored by a simple link-time check. This partial order among classes can also help with specifying and verifying frame conditions of static fields.

In this variation of the Boogie methodology, a class is in one of the states $mutable$, $valid$, or transitively-valid ($tvalid$). In the context of invariants over static fields, $mutable$ means that the class is in a state in which its static fields may be assigned. If a class is in state $valid$ or $tvalid$, its invariant is known to hold. The methodology guarantees the following property at all times: if a class is in state $tvalid$, then so are all the classes that it succeeds in the class ordering. The typical precondition of a method is that the enclosing class is in state $tvalid$, which means that the invariant of the enclosing class and all the classes that it succeeds in the class ordering may be assumed to hold.

A possible frame rule that governs the modification of static fields is: a method is allowed to affect the static fields of any class that, in the pre-state of the method, is not in state $mutable$ [88]. For example, since class `Operation` in Fig. 9 on page 15 is a client of class `Date`, `Operation` would be declared to succeed `Date` in the class ordering. The precondition of `perform` would say that class `Operation` is in state $tvalid$, which then implies that class `Date` is also in state $tvalid$. Since method `now` does not rely on the invariant of `Operation`, it would only require that `Date` be in state $tvalid$. This allows the implementation of `now` to meet the precondition of `Date.getTime`, namely that `Date` is in state $tvalid$.

In order to mutate the state of a class, the methodology says that the class has to be in state $mutable$. Thus, `perform` would need to change `Operation` into state $mutable$ before assigning to the fields `operations` and `elapsedTime`. Because `Operation` succeeds `Date`, this has no effect on the state of `Date`. Now the proposed frame condition comes into play: because `Operation` is in state $mutable$ during the executions of `now` and `Date.getTime`, the frame rule says that the static fields of `Operation` are unchanged. This allows the implementation of `perform` to be verified.

But this frame rule is not entirely satisfactory, because it says nothing about the static fields of classes not in state $mutable$. Imagine that method `now` would require class `Operation` to be in state $tvalid$. Then method `perform` would make `Operation` $mutable$ only between the calls to `now` instead of during the execution of the whole method body. Consequently, `now` would be allowed to modify static fields of `Operation` without declaring these modifications in its assignable clause, and we could not verify `perform`. In this particular example, since `now` is a method of class `Operation`, one could add a stronger ensures clause to express that it does not modify static fields of `Operation`. However, this would in general not be possible if the method was declared in a different class. One could imagine making the frame rule stricter, to say that a method in a class C can only ever have an effect on the static fields in classes that C succeeds. However, this does not help if the relative order of two classes is unknown.

3.3 Class Initialization

Modern programs often rely on large library components. Reducing the time to load and initialize these libraries is important to improve the responsiveness of the programs. A solution employed by both the Java Virtual Machine and

the .NET Common Language Runtime is to support lazy class initialization. This means that a class is not loaded and initialized until its first use. While this feature can improve responsiveness, it complicates reasoning about programs.

Consider the following program snippet:

```
int x = A.a;
int y = B.b;
int z = A.a;
assert x == z;
```

where `A.a` and `B.b` are static fields in two classes, `A` and `B`. If one expects the reading of static fields to have no side effects, then one would conclude that the assertion will hold. However, given the following class declarations:

```
class A {
  public static int a;
  static { a = 0; }
}
class B {
  public static int b;
  static { b = 0; A.a = 5; }
}
```

it is possible, in the presence of lazy class initialization, to arrive at the assertion with local variables `x` and `z` having the values 0 and 5, respectively. This would happen if class `B` has not yet been loaded at the time `B.b` is read; the reading of `B.b` will then first invoke class `B`'s static initializer, which sets `A.a` to 5.[4]

We certainly need to allow static initializers to mutate state, but it would be horribly non-modular to have to reason about the possibility of such mutations happening any time something from another class is referenced. What we would like is a specification and verification technique that confines the side effects of class initializers in such a way that one can ignore the specific time at which they actually occur.

Challenge 5 *Develop a specification and verification technique for lazy class initialization.*

3.3.1 Solution Approach 1: Limit Class Initialization Invocations

In some programming systems, it may be possible to limit when class initializers are invoked. For example, the .NET Common Language Runtime allows some flexibility in when class initialization takes place. If class initializers are invoked only when the program reaches a procedure boundary (call or return), then it may be possible to extend a solution to Challenge 4 to also account for the state being modified as part of class initialization. A solution along these lines would still need to develop restrictions on what state static initializers are allowed to modify.

3.3.2 Solution Approach 2: Class Ordering

Reasoning about eager class initialization, including the eager initialization of dynamically loaded classes and libraries, can be facilitated by a partial order on the classes as described in Sec. 3.2.2. That methodology prescribes when it is possible to rely on invariants declared about static fields [88]. Perhaps it is possible to use the class ordering to restrict the modifications of static initializers to those static fields declared in predecessor classes. By doing so, it seems possible to regain the property that reading a static field leaves the program's state unchanged.

Note that it is not sufficient simply to define a rule that a class may only assign to static fields declared in superclasses. For instance, if class `B` in the example above were declared to be a subclass of `A`, then the same problem would exist. Also, note that it is not sufficient simply to define a rule that says a class can only assign directly to its own static fields. In the example above, we could replace `B`'s direct assignment to `A.a` by a call to a method in `A` that would do the actual assignment to `A.a`, in which case the problem would still exist.

The two solution approaches we have alluded to here may, at best, hint at some ingredients that may be useful in solving Challenge 5. A full solution remains an open challenge.

[4] This problem has been noted by others. For example, N. G. Fruja mentioned it in his talk at the .NET Technologies workshop in Plzeň, Czech Republic, in June 2004.

4 Heap Data Structures

Over the last decade, research in program verification has seen tremendous improvements in reasoning about heap structures and aliasing [13, 87, 90, 101, 102, 103, 105, 108, 111]. In this section, we discuss two of the remaining challenges, the verification of invariants of complex object structures and the verification of finalizers.

4.1 Invariants of Complex Object Structures

Almost all interesting data structures consist of several interacting objects. The invariant of such a data structure relates the state of several objects, which implies that modifications of these objects potentially affect the invariant. Consequently, a sound verification technique has to generate proof obligations for all methods that modify an object of the data structure to maintain the invariant.

We illustrate invariants of object structures by the implementation of the Composite Pattern [50] in Fig. 10 on the next page. Each `Composite` object stores references to its direct sub-components in an array. The invariant of a `Composite` object x expresses that the field x.`total` contains the number of components of the composite tree rooted in x. Therefore, the invariant of x depends on the state of x, the array x.`components`, and the state of x's direct sub-components. Any method that modifies any of these objects potentially violates x's invariant. Therefore, the invariant leads to proof obligations for the methods of `Component` and its subclasses, methods of `Composite`, and any method that has access to the `components` array. To handle the latter group of methods in a modular way, one has to employ some form of alias control to limit this group, for instance, to the methods of `Composite`.

Our example illustrates that reasoning about invariants of object structures has to deal with the complications of subclassing and aliasing, and is, therefore, a rather difficult challenge:

Challenge 6 *Develop a specification and verification technique for invariants of complex object structures.*

4.1.1 Solution Approach 1: Ownership-Based Invariants

Ownership [32] provides encapsulation for object structures, which can be used to verify invariants modularly [87, 101, 103]. Ownership organizes a data structure hierarchically into an interface object (the owner), which is used by clients to access the data structure, and representation objects, which are accessed only via their owner. An admissible ownership-based invariant of an object x depends only on fields of x and objects owned by x. Therefore, x has full control over any modifications that potentially affect the invariant, and a verification methodology can impose appropriate proof obligations on the methods of x.

Although many object structures are well encapsulated and can be verified using ownership, there are several practically relevant data structures that expose their objects to clients. For instance, implementations of the Composite pattern typically do not encapsulate the sub-components of a composite. Clients can add components to any composite of the hierarchy, not only to the root composite. Forcing clients to always access a hierarchy through its root r would be highly inefficient because the add operation would have to determine the roots of all sub-hierarchies between r and the composite where the new sub-component should be added. In general, this requires a costly traversal of the hierarchy. Consequently, a `Composite` object does not own its sub-components, and the invariant of class `Composite` is not an admissible ownership-based invariant. The invariant is nevertheless maintained because the `addComponent` method triggers a bottom-up traversal of the composite structure to re-establish the invariant. In our example, this traversal is done by method `addToTotal`, which adjusts the `total` fields of the (transitive) parent composites.

In summary, ownership-based verification is a powerful technique that is useful for many practical examples, in particular, aggregate objects. However, some heap data structures such as the Composite pattern maintain interesting invariants, but do not follow an ownership discipline [13]. Therefore, ownership-based invariants are only a partial solution to Challenge 6.

4.1.2 Solution Approach 2: Visibility-Based Invariants

While ownership-based invariants gain their modularity from a strong encapsulation, visibility-based invariants [13, 87, 101, 103] gain their modularity from enforcing that all invariants that are potentially affected by a field update are visible in the method that contains the update. Therefore, it is possible to show modularly that these invariants are preserved. In our example, we assume that the classes `Component` and `Composite` are declared in the same

```
class Component {
  protected /*@ spec_public @*/ Composite parent;
  protected int total = 1;

  //@ protected invariant 1 <= total;
}

class Composite extends Component {
  private Component[] components = new Component[5];
  private int count;

  //@ private invariant total == 1 + (\sum int i; 0 <= i && i < count; components[i].total);

  //@ requires c.parent == null;
  public void addComponent(Component c) {
    // resize array if necessary
    components[count] = c;
    count++;
    c.parent = this;
    addToTotal(c.total);
  }

  //@ requires 0 <= p;
  private /*@ helper @*/ void addToTotal(int p) {
    total += p;
    if(parent != null) { parent.addToTotal(p); }
  }
}
```

Figure 10: An implementation of the Composite pattern. As expressed by the JML invariant in `Composite`, the `total` field stores the number of (sub-)components of a component. For simplicity, we omit the invariants that express that each component is correctly linked to its parent. The invariant shown in `Composite` uses `\sum` to express the addition of all component totals. Method `addToTotal` is declared as `helper`, which means that it cannot assume and need not preserve any invariants. The `helper` and `private` modifiers were missing in the published version of this article, but are required for the example to be correct.

```
import java.io.*;

public class TempStorage {
  private /*@ nullable @*/ FileReader tempFile;
  private /*@ nullable @*/ FileWriter logFile;

  //@ private invariant tempFile != null && logFile != null;

  public TempStorage() throws IOException {
    tempFile = new FileReader("/tmp/dummy");
    logFile  = new FileWriter("/tmp/log");
  }

  protected void finalize() throws Throwable {
    super.finalize();
    logFile.write("Bye bye");
    logFile.close();
    tempFile.close();
  }
}
```

Figure 11: The `finalize` method closes the files used by the receiver object. Although non-null is the default in JML, we include, for emphasis, a declaration that makes this invariant explicit.

package and, therefore, mutually visible. In particular, `Composite`'s invariant is visible in every method that updates `Component`'s `total` field. Therefore, it is possible to generate proof obligations that these methods maintain the invariant.

Visibility-based invariants are useful for data structures that do not follow an ownership discipline. However, they have several severe drawbacks. First, the visibility requirement is often too strict. For instance, visibility-based invariants must not depend on array elements because every method in a program that gets hold of a reference to an array can modify it. Without alias control (such as ownership), the set of such methods generally cannot be determined modularly. Second, the visibility requirement does not support subtyping well. For instance, the invariant of a subclass of `Component` in a different package cannot refer to the `total` field because this invariant is not visible where the `total` field is declared. If the subclass invariant could refer to `total`, then methods in `Component`'s package could break this subclass invariant by assigning to `total`. However, since the subclass invariant is not visible in these methods, they cannot be required to maintain it. Third, visibility-based invariants increase the number and complexity of proof obligations. For instance, the fact that the composite data structure forms a tree is trivial if composites own their sub-objects (since ownership is a tree order) but has to be specified and verified explicitly if no ownership discipline is applied. Such a specification involves reachability predicates, which are difficult to reason about, especially by automatic theorem provers [40].

Due to these drawbacks, visibility-based invariants are useful to complement ownership-based invariants, but cannot replace them. The invariant of the composite example in Fig. 10 on the preceding page can be expressed using a combination of ownership (for the `components` array) and visibility (for `Component-Composite`). However, this combination still suffers from the second and third drawback. Complex heap structures such as the Composite pattern require new solutions to Challenge 6.

4.2 Finalizers

Finalizers are special methods that are invoked by the runtime system before an unreachable object is de-allocated. Their purpose is mainly to free system resources. For instance, the `finalize` method in Fig. 11 closes the files used by its receiver object.

Since the runtime environment of languages like Java and C# may invoke the garbage collector in any execution

state, programs have no control over the execution of finalizers. This leads to two problems for verification.

First, a finalizer might be invoked in a state in which certain object invariants do not hold. In our example, the constructor of `TempStorage` throws an exception if opening the files fails. In this case, the object is never fully initialized and thus its invariant does not hold. However, the finalizer of `TempStorage` relies on the object invariant and, therefore, will abort with a null-pointer exception when a partly-initialized object is destroyed. A verification technique can prevent an application program from calling a method on a partly-initialized object, for instance, by making explicit which object invariant may be assumed to hold [10]. However, finalizers are called by the runtime system and, therefore, cannot be controlled.

Second, like any other method, a finalizer potentially modifies the heap. Since finalizers might be called in any execution state, a verification technique has to deal with spontaneous heap changes, which is even worse than the heap changes caused by static initializers (see Sec. 3.3).

Dealing with these problems is an open challenge:

Challenge 7 *Develop a verification technique for finalizers.*

A solution to this challenge is necessary to guarantee that verification is not unsound for programs containing finalizers.

4.2.1 Solution Approach: Severe Restrictions

Since the runtime system may invoke finalizers in any execution state, reasoning about finalizers is similar to reasoning about multi-threaded programs. However, multi-threading is very general, whereas finalizers are mainly used for the special purpose of freeing system resources. Therefore, a specification and verification technique may impose strong requirements on finalizers that would be too restrictive for multi-threading.

To deal with the first problem, we do not see an alternative to simply not making any assumptions about the heap in finalizers. Any property a finalizer requires has to be checked at runtime. For instance, the method invocations in our example have to be guarded by checks that the corresponding receivers are non-null. In order to allow finalizers to call methods of other objects, which typically require their invariants to hold, it would be helpful to allow programs to explicitly check at runtime whether certain invariants hold.

Concerning the second problem, it seems necessary to allow a finalizer to modify only those objects and system resources that are exclusively used by the object that is being destroyed. In particular, finalizers must not modify global state such as static fields. Techniques such as ownership type systems may be useful for reasoning about the sharing of objects. However, it is unclear how to guarantee that certain system resources are not shared, for instance, how to prevent two objects from creating handlers for the same file.

5 Control Flow

Program logics that can handle jumps [8, 11, 16, 37, 65] solve the earlier verification challenges of dealing with unstructured control flow such as abrupt termination of loops and exceptions. However, a remaining challenge is to deal with higher-order features, for instance, a filter method that takes a reference to a predicate method that determines whether a data element should be filtered out of a collection. Higher-order features occur in object-oriented programs in the form of objects that act as functions, which we refer to as *function objects*.

A type-safe way of implementing function objects in object-oriented languages is the Strategy pattern [50]. This pattern consists of an interface with the signature of the method that should be passed as a function object and subclasses implementing this method. Alternatively, C#'s delegates [42] and Eiffel's agents [43] are language features that provide type-safe function objects. In this section, we discuss how to specify methods that use function objects and how to verify invocations of function objects. We illustrate the challenges by the Strategy pattern and delegates with a single underlying method, but the discussion also applies to multicast delegates and Eiffel agents.

5.1 Specification of Methods that Use Function Objects

Fig. 12 on the following page shows a typical application of function objects. In this example, the `format` method of class `Paragraph` takes a delegate as an argument. This delegate represents a format algorithm that is applied to the text paragraph. Class `Formatters` provides two implementations of formatters that can be used to instantiate

```
class Paragraph {
  char[][] text;
  int width;

  //@ assignable text;
  public void format(Formatter f) {
    f(this);
  }
}

//@ assignable p.text;
delegate void Formatter(Paragraph p);
```

```
class Formatters {
  //@ assignable p.text;
  public static void alignLeft(Paragraph p)
  { /* modify p.text */ }

  //@ assignable p.text;
  public static void alignRight(Paragraph p)
  { /* modify p.text */ }
}
```

Figure 12: An implementation of text paragraphs with two formatters. The formatter for a text paragraph is passed to the `format` method as a delegate. In this example, we have used Java and JML notation extended with C# delegates.

```
class Paragraph {
  /*@ spec_public @*/ char[][] text;
  int width;

  //@ assignable text;
  public void format(Formatter f) {
    f.formatParagraph(this);
  }
}

interface Formatter {
  //@ assignable p.text;
  void formatParagraph(Paragraph p);
}
```

```
class AlignLeft implements Formatter {
  //@ also assignable p.text;
  public void formatParagraph(Paragraph p)
  { /* modify p.text */ }
}

class AlignRight implements Formatter {
  //@ also assignable p.text;
  public void formatParagraph(Paragraph p)
  { /* modify p.text */ }
}
```

Figure 13: An alternative implementation of the example in Fig. 12 using the Strategy pattern (in Java and JML).

the delegate `Formatter`. The formatters format the text by directly modifying the `text` array of the `Paragraph` object `p`. Fig. 13 shows the example using the Strategy pattern instead of a delegate.

Since different format algorithms can have very different behavior, we cannot completely specify their effect in an ensures clause of the `Formatter` delegate (or the `formatParagraph` method of the interface `Formatter` of Fig. 13). This is typical for function objects. The various methods a function object can be instantiated with often have almost no common behavior that could be described in a specification of the function object. For instance, the update methods of different observers in the Observer pattern may react to an event completely differently. This distinguishes function objects from virtual methods with overriding implementations, where typically all implementations share some common behavior.

It is also not possible to give a direct and complete specification of the effect of `format` on a `Paragraph` object. Verifying such a specification would require knowledge about the effect of the invocation of the function object, but this knowledge is not available because the function object does not (and cannot) have a strong specification.

Since we cannot give a direct specification for `format`, we would like to specify its behavior relative to the behavior of the function object. In other words, we would like to specify that `format` applies the delegate instance `f` (or the method `f.formatParagraph`) to its receiver object. However, to use `f` in a mathematical description of the behavior of `format` one must summarize `f`'s behavior mathematically. This brings us again to the problem of using

method calls in specifications. In JML, using `f` directly in a specification would require that the delegate `f` be pure, and hence that all methods the delegate can be instantiated with also be pure. But this is not the case in our example since these methods modify the `text` array. Function objects that are non-pure methods are common and occur, for instance, in the cooperation between containers and layout managers in the Java Abstract Windowing Toolkit. Thus reasoning about function objects is an interesting research challenge:

Challenge 8 *Develop a specification technique for methods that use function objects.*

5.1.1 Solution Approach 1: Pure Methods

Our first solution approach works for function objects that contain pure methods, which may be used in specifications. Recent work on the encoding of pure methods [34, 38, 69] can be generalized to function objects. These encodings introduce a mathematical function for each pure method of a program. The functions for pure methods are axiomatized based on the method specifications.

A possible encoding of pure delegates is a mapping from delegate objects to the functions for the underlying pure methods. However, such a second-order encoding is not supported by automatic theorem provers like Simplify [40]. Therefore, one has to develop alternative encodings in first-order logic.

This approach allows us to specify and verify applications of function objects to pure methods. However, this approach does not work for non-pure methods like in the `Paragraph` example (Fig. 12 on the previous page and Fig. 13 on the preceding page) because these methods cannot be used in specifications.

5.1.2 Solution Approach 2: Pure Check Methods

Based on the previous approach, we can handle non-pure function objects by always passing pairs of methods, the non-pure method we want to call and a pure boolean check method that simply checks the effect of the non-pure method. Objects containing such pairs of methods are easily implemented in the Strategy pattern, where we can simply add a second method to the interface.

Fig. 14 on the next page illustrates this approach. In addition to the impure method `formatParagraph`, interface `Formatter` declares a pure check method `isFormatted`, which is used in the specification of `format`. The formal connection between the method `formatParagraph` and its check method `isFormatted` is the ensures clause of `formatParagraph`.

Now we can use the first approach (Sec. 5.1.1) to verify the example. Provided that we know the concrete type of the `Formatter` object passed to `format`, we can use the specification of the check method in that type to reason about the effects of `format`.

This approach is a partial solution to Challenge 8 for the Strategy pattern. Unfortunately, it increases the specification overhead because pure check methods have to be added. Moreover, it requires a solution to Challenge 3 to be useful in practice. It is not immediately clear how to extend this approach to delegates, which do not support the pairwise combination of a method and its check method.

5.1.3 Solution Approach 3: Specification Reflection

C#'s delegates, Eiffel's agents, and the Strategy pattern allow a specification language to equip function objects with specifications and purity annotations. The previous two solution approaches show how to use such specifications for static verification. Function objects can also be implemented by reflection, for instance, using class `Method` in Java. This solution is not type-safe and does not allow one to associate specifications with function objects. It may be possible to extend the reflective capabilities of the language to also allow access to specifications [29], or to extend the specification language to permit access to the specifications of such objects [72]. However, the details of how to do static verification with such specifications remain to be worked out.

5.1.4 Solution Approach 4: Model Programs and Enriched Traces

Yet another approach to this challenge is based on ideas of the refinement calculus [6, 100]. The "greybox" approach to specification of Büchi and Weck [20, 21] specifies such higher-order procedures using abstract programs. JML's "model programs" are designed to allow for specifications in this style. As an example, the specification of method `format` from Fig. 14 on the following page could be given as follows.

```
class Paragraph {
  /*@ spec_public @*/ char[][] text;
  int width;

  //@ assignable text;
  //@ ensures f.isFormatted(this);
  public void format(Formatter f) {
    f.formatParagraph(this);
  }
}

interface Formatter {
  //@ assignable p.text;
  //@ ensures    isFormatted(p);
  void formatParagraph(Paragraph p);

  /*@ pure @*/
  boolean isFormatted(Paragraph p);
}
```

```
class AlignLeft implements Formatter {
  //@ also
  //@   assignable p.text;
  public void formatParagraph(Paragraph p)
  { /* modify p.text */ }

  //@ also
  //@ ensures (* \result == p is left aligned *);
  public /*@ pure @*/
  boolean isFormatted(Paragraph p)
  { /* ... */ }
}


class AlignRight implements Formatter {
  //@ also
  //@   assignable p.text;
  public void formatParagraph(Paragraph p)
  { /* modify p.text */ }

  //@ also
  //@ ensures (*\result == p is right aligned*);
  public /*@ pure @*/
  boolean isFormatted(Paragraph p)
  { /* ... */ }
}
```

Figure 14: A JML specification of the example in Fig. 13 on page 23 using pure check methods. In JML, `(*` and `*)` delimit informal specifications.

```
//@ public model_program { f.formatParagraph(this); }
public void format(Formatter f) { f.formatParagraph(this); }
```

The model program in this case is quite simple, and just exposes the essential form of a conforming implementation to the clients. So in this case, there are essentially no other correct implementations. In more interesting cases, one can use specification statements to leave parts of such a method up to the implementation. Details of the semantics of model programs in JML remain to be worked out (especially how to verify that a method satisfies the specification of such a model program [113]). While the model program in theory contains enough information to reconstruct a trace of the program's execution, the technique by itself does not solve the challenge, because it does not provide a direct way for clients to verify interesting properties about calls to methods that have model program specifications.

This challenge is more directly addressed by the work of Soundarajan and Fridella [114]. In their work, specifications for function objects have an additional part, called an "extended specification". The extended specification describes what traces of method calls may result from the method's execution. These traces allow clients to derive stronger constraints on the post-state, by plugging in (more) exact specifications for the method calls in the trace. That is, the extended specification is parameterized on the meaning of the methods it calls; if the client knows more about such methods, then this extra knowledge can be used to strengthen what is known about the post-state. While reasoning using extended specifications and traces is not simple, it seems like a promising direction for this challenge. Soundarajan and Fridella claim both soundness and a kind of completeness for their technique.

5.2 Verification of Invocations of Function Objects

In the previous subsection, we discussed how to specify methods that use function objects. In this subsection, we focus on a related problem, namely how to prove that the invocation of a function object is correct.

Fig. 15 on the next page shows an implementation of a storage system. The `Archive` delegate allows one to create function objects for the store methods of different archives. In method `Main`, the `Archive` delegate is instantiated with the instance method `store`. As illustrated by this example, instantiation of a delegate with an instance method also fixes the receiver object of calls to this method, in this case, `tapeArchive`.

Invoking a function object triggers a call to the underlying method. Verification has to ensure that the requires clause of this method holds when the function object is invoked. Conversely, the properties guaranteed by the underlying method should be available at the invocation site. The challenge is to enable this kind of reasoning.

Challenge 9 *Develop a specification and verification technique for function objects.*

5.2.1 Solution Approach 1: Pre-Post-Specifications and Refinement

With the Strategy pattern, invocations of function objects are verified using the specification of the method in the Strategy interface. Behavioral subtyping enforces that all implementations of this Strategy method refine its specification.

To adapt this approach to delegates, we associate each delegate declaration with a specification similar to method specifications. When a delegate type D is instantiated with a method m, one has to prove that m's specification refines D's specification. More precisely, one has to prove that D's requires clause is stronger than m's and that D's ensures clause is weaker than m's when D's requires clause holds. At the invocation site of the delegate, it suffices to prove that the requires clause of D holds, which implies that the weaker requires clause of m holds as well. Conversely, one may assume D's ensures clause after the invocation.

Ignoring for the moment the requires clause of method `store`, which will be used in a later example, the delegate `Archive` and the method `store` have identical requires and ensures clauses. Therefore, `store`'s specification trivially refines the specification of `Archive`, which allows us to verify the delegate instantiation in method `main`. When the delegate is invoked in method `log`, we have to prove that the requires clause of the delegate is satisfied, which in this example also is trivial.

Equipping delegates with specifications and checking a refinement relation when a delegate is instantiated allows us to verify many delegate invocations such as the example in Fig. 12 on page 23. However, this approach is insufficient when a delegate is instantiated with a method whose specification refers to properties of the receiver object. The problem is illustrated in Fig. 15 on the next page by the requires clause of `store`, which requires the model field `isReady` of the receiver to be true. In order to ensure that the specification of `store` refines the specification of `Archive`, `Archive`'s requires clause has to express properties of the receiver of the underlying method. This can be done using the `_target` field of C#'s `Delegate` class:

```
class Tape {
  public void save(Object o)
  { /* ... */ }

  // other methods omitted
}

class TapeArchive {
  /*@ nullable @*/ Tape tape;

  //@ public model boolean isReady;
  //@ represents isReady
  //@             <- tape != null;

  //@ ensures isReady;
  public TapeArchive()
  { tape = new Tape(); }

  //@ requires isReady;
  public void store(Object o)
  { tape.save(o); }

  //@ requires isReady;
  //@ ensures  !isReady;
  public void eject()
  { tape = null; }
}
```

```
delegate void Archive(Object o);

class Client {
  public static void log(Archive logfile,
                         String s) {
    logfile(s);
  }
}

public class Main {
   public static void main(String[] args) {
    TapeArchive tapeArchive = new TapeArchive();
    Archive archive =
       new Archive(tapeArchive.store);
    Client.log(archive, "Hello World");
  }
}
```

Figure 15: A implementation of a tape archive and its client. The **`represents`** clause says that the model field `isReady` is true when a tape is loaded in the archive. The model field provides an implementation-independent specification for the methods of `TapeArchive`. The delegate `Archive` provides clients with a uniform way of storing data in different archives.

```
interface ArchiveStrategy {
  void apply(Object o);
}

class TapeArchiveAdapter
      implements ArchiveStrategy {

  /*@ spec_public @*/ TapeArchive ta;
  //@ invariant ta.isReady;

  //@ requires t.isReady;
  public TapeArchiveAdapter(TapeArchive t) {
    ta = t;
  }

  public void appy(Object o) {
    ta.store(o);
  }
}
```

```
class Client2 {
  public static
  void log(ArchiveStrategy logfile,
           String s) {
    logfile.apply(s);
  }
}

class Main {
   public static void main(String[] args) {
    TapeArchive tapeArchive =
      new TapeArchive();
    ArchiveStrategy archive =
      new TapeArchiveAdapter(tapeArchive);
    Client2.log(archive, "Hello World");
  }
}
```

Figure 16: An implementation of the storage example based on the Strategy and Adapter patterns.

```
  /*@ requires _target != null && _target is TapeArchive
    @       ==> ((TapeArchive)_target).isReady;
    @*/
```

With the appropriate substitution, it is trivial to show that this requires clause implies the requires clause of method `store`. However, the above requires clause entails two problems. First, using `_target` in the delegate specification requires callers of the delegate to reason about properties of the receiver object. This is cumbersome because these properties have to be propagated from the instantiation of the delegate (where the receiver is known) to each invocation site. For instance, we have to add a similar requires clause to method `log` to verify the delegate invocation. Second, the specifier of `Archive` has to foresee that the delegate might be instantiated with a method of `TapeArchive`. Otherwise, they would not specify a requires clause for `Archive` that accesses `isReady`. This deprives delegates of much of their flexibility. In particular, adding a new method such as `DiskArchive.save` to the program requires an additional requires clause for `Archive`, which cannot be added without changing the existing code.

These problems are avoided by an implementation using a simple Strategy pattern instead of delegates. The model field `isReady` could then be declared in the Strategy interface and used in the specification of the Strategy method. Different subclasses of the Strategy interface can provide different representations for the model field. However, such a simple Strategy pattern requires that all implementations of the Strategy method have the same name and be declared in subclasses of the Strategy interface, which is often too restrictive. These restrictions are eliminated when the Strategy pattern is combined with an Adapter pattern. We discuss an approach for this design next.

5.2.2 Solution Approach 2: Visibility-Based Invariants

The code in Fig. 16 shows a pattern-based implementation of the storage example from Fig. 15 on the previous page. The Strategy interface `ArchiveStrategy` declares the method `apply`, which is used to invoke the function object. To achieve the same flexibility as with delegates, in particular, to be able to instantiate the function objects with methods with different names or from classes that do not implement `ArchiveStrategy`, we combine the Strategy with an Adapter pattern. Class `TapeArchiveAdapter` is the adapter for class `TapeArchive`. It delegates invocations of `apply` to the `store` method of the `TapeArchive` instance `ta`. A similar adapter would be needed to instantiate the function object with a method `DiskArchive.save`.

In this pattern-based implementation, the receiver for an invocation object is stored in a field of the adapter object. Therefore, properties of the receiver object can be expressed as object invariant of the adapter as shown in class

`TapeArchiveAdapter`.

Let's assume a visible state semantics for invariants [57, 98], where all object invariants hold in the pre- and post-states of all method executions. In our example, we can prove that the instantiation of `TapeArchiveAdapter` in method `main` satisfies the requires clause of the constructor, which establishes the invariant. The visible state semantics allows us to assume that the invariant of `logfile` holds in the pre-state of method `log` and, therefore, to verify the invocation of the delegate. The verification of `log` neither needs additional requires clauses nor involves properties of the receiver of the function object. This shows that invariants solve the first problem of solution approach 1 (Sec. 5.2.1). The second problem of solution approach 1 is solved because the invariant is declared in the adapter class, not the Strategy interface. In our example, the implementor of the interface `ArchiveStrategy` does not have to foresee that the function object will be instantiated with `TapeArchive.store`.

If the invariant of a function object refers to the state of the receiver of the underlying method, it can be violated by modifying this receiver. Suppose method `main` in Fig. 16 on the preceding page calls `tapeArchive.eject()` before calling `Client2.log`. The call to `eject` violates the invariant of the adapter `archive`. Therefore, the extended example should not verify. As discussed in Sec. 4.1, existing work provides two modular verification techniques for invariants of object structures.

Ownership-based invariants require the adapter object to own the receiver of the underlying method. They prevent the call `tapeArchive.eject()` because ownership forces all accesses to an owned object to be initiated by the owner, in this case, `archive`. This is clearly too restrictive for many programs. Consider for instance an implementation of the model-view-controller architecture where the controller uses function objects to dispatch events to the model. Using ownership would mean that the model can be accessed only through callbacks from the controller, which is not realistic. Moreover, existing ownership systems support only single ownership. Therefore, the receiver of a function object could not be part of another ownership hierarchy.

Visibility-based invariants are better suited for function objects. The invariant of `TapeArchiveAdapter` is an admissible visibility-based invariant if `TapeArchiveAdapter` is visible in class `TapeArchive`, for instance, because both declarations are contained in the same package. The visibility of the invariant allows us to impose proof obligations that each method that modifies `isReady` preserves the invariant. The specification in Fig. 15 on page 27 does not allow one to show this proof obligation for method `eject`.

Visibility-based invariants provide a partial solution to Challenge 9, but leave some problems unsolved. First, they require the adapter to be declared in the same package as the fields mentioned in its invariant. In particular, it is not possible to declare an adapter with an invariant that mentions a field from a library class because in this case, the adapter class is not visible where the field is declared. This is a severe restriction on reuse. Second, with a visible state semantics, invariants have to be preserved by all methods of a program. In our example, the call x`.eject` violates the invariant of any `TapeArchiveAdapter` object that references x. Consequently, `eject` needs a requires clause that no such `TapeArchiveAdapter` object exists. In a pattern-based implementation, this requirement can be established by setting the `ta` field of all relevant `TapeArchiveAdapter` objects to null. However, delegates do not provide such an operation to detach their target. Barnett and Naumann [13] present a powerful methodology for dealing with visibility-based invariants, but adapting their methodology to the peculiarities of delegates has not yet been attempted.

6 Practical Considerations

In addition to the technical specification and verification challenges described above, there are also challenges of a more practical nature. These involve the ease of using the specification language, its expressiveness, and tool support for both specification and verification.

6.1 Library Specifications

One of the most important and difficult practical problems is obtaining specifications of standard class libraries, such as the libraries that come with Java and C#. These libraries are especially important for the verification of real programs, since most programs make heavy use of them; hence calls to methods in such libraries can only be verified if the libraries are specified. For example, if the assignable clause for a library method m is not given, then callers of m have to assume conservatively that m modifies the state of all reachable objects. In particular, m might be overridden in subclasses such that a call to m may modify even fields that are not accessible to the library implementation of m.

Furthermore, library methods are usually called to achieve some particular postcondition, so fairly complete functional specifications will be needed by many clients.

The sheer size of the libraries that come with C# and Java makes the task of writing functional specifications for these libraries daunting and costly. It would be ideal if the designers of these libraries had written specifications for them, but failing that, some automated inference of specifications can be very helpful in making this task more practical. Of course, an automated inference process cannot precisely infer design intent, so some decisions, for example about preconditions, will need human judgment. Moreover, human input will be needed to decide what are the appropriate abstractions. However, if a person decides what an appropriate abstraction would be for a type, then it may be possible to automatically infer a reasonable specification (or set of likely specifications for many cases).

Challenge 10 *Provide assistance in specifying libraries of classes.*

One tool that can help with creating specifications is Daikon [22, 45]. It can infer specifications by performing data mining on information gathered from test runs. However, using it requires a test suite that will exercise the relevant modules. A similar approach, but without the requirement of a test suite, was taken by Houdini [48], which guessed various method specifications and used ESC/Java [49] to prune away invalid guesses. Nimmer and Ernst worked to combine these approaches by using ESC/Java to prune invariants produced by Daikon that could be statically shown to be invalid [104].

However, all of these efforts produce specifications that describe the effects of methods on (private) fields. Allowing users to specify abstractions, or inferring them, remains a challenge.

6.2 Dealing with Multiple Tools

Specifiers often write specifications in different styles and at different levels of detail and completeness. One reason for this is that they may only be interested in using certain tools (such as a runtime assertion checker or static checker) and not others. Having several different kinds of tools able to work on specifications is a benefit, as different tools have different strengths and weaknesses [22]. However, having a choice of tools means that documentation is needed that explains what features of the specification language are relevant for each of the tools. For example, unbounded quantifiers are not usually considered executable by runtime assertion checking tools.

To design a specification language that serves the needs of several tools is thus a balancing act. The semantics of the specification language has to be designed to work with all the different kinds of tools. Fortunately, it seems that the needs of runtime checking and static verification are not incompatible [79]. Nevertheless, besides needing more complete specifications, static verification often needs extra specification constructs, such as intermittent assertions and assumptions, loop invariants or specifications of the entire effect of a loop [58, 59], and axioms. Different static verification systems may also have different needs, for example based on their different strategies for handling loops and recursion. This leads to the following challenge.

Challenge 11 *Carefully document what specification language constructs are useful for which tools, and make sure the semantics of all these constructs are compatible.*

One approach to organizing documentation that may be helpful to users is to specify a graduated sequence of language subsets. For example, one might specify a subset for runtime assertion checking, a larger one for extended static checking, and a yet larger one for formal verification. This would also help users understand what constructs are useful for what tools. The Omnibus environment takes such an approach and offers suggestions for how different specification styles can be combined [119]. Yet another way to organize the documentation might be based on which features are most often needed.

A related practical problem is that when one first starts using a new feature of a specification language, it often must be used everywhere at the same time. For example, if one adds an assignable clause to a method M, one must specify assignable clauses for all methods that M calls, and then all methods called by those methods, etc. Tools might helpfully point out a (bottom-up) ordering that would allow useful checking during intermediate phases of such additions.

Another related practical problem is to hide the complexity of various verification techniques from users by suitably chosen defaults. One would like the flexibility to override such defaults when necessary, but suitable defaults can greatly ease the burden of writing and reading specifications in the normal case, especially if the need to override the defaults is rare.

In our experience with Spec#, we have found it awkward to override the defaults, because it is difficult to be specific about which part of the default should be overridden. For example, the current default of an ordinary method in Spec# is that all parameters are "peer consistent" on entry and that the return value is "peer consistent" on exit. If one wants a different precondition for, say, the receiver parameter, then one would mark the whole method as not getting any default specifications, which means that the default specifications for the other parameters must all be provided explicitly. The rationale behind this design of providing only one way to turn off (all) default specifications for a method was to simplify the way a user works with defaults, but this has not worked out so smoothly. Perhaps some support from an integrated development environment (IDE) could help here, since an IDE might be able to, upon request by a user, display the defaults and allow them to be explicitly deleted or changed. The IDE could still keep the defaults from view in the common cases.

7 Conclusions

In this paper, we described specification and verification challenges that we currently face in our work on JML and Spec#. In trying to write down various challenges and solution approaches, we also found a few problems that turned out to already have solutions. Two of the more notable of these are the following:

- Supporting field-like properties of objects, which is solved by using model fields [30, 84]
- Specifications of coroutine-like iterators, which is solved by the techniques of Jacobs *et al.* [68].

For many of the remaining challenges, we could at least give partial solutions or describe promising approaches. We hope that the identification and description of the remaining challenges will help increase the understanding of some important issues in specification and verification of sequential object-oriented programs, and that the described approaches will help point out some likely avenues for future research.

Our list of challenges is not complete. For example, we did not consider how to specify and verify programs that use multi-threading, reflection, dynamic class loading, and other advanced features. There are certainly plenty of challenges in these and other areas. Furthermore, in the short period of time between our paper's submission and the final version, we had to rewrite several sections to accommodate the latest results; this shows how active the field is. Nevertheless, we hope that our list of challenges and their analysis will help, in a small way, the field make progress towards the grand challenge of verified software.

Acknowledgments

We are grateful to Kristina Boysen, David Cok, Ádám Darvas, Fraaz Hussain, Bart Jacobs, Joseph Kiniry, and Joseph Ruskiewicz for their comments on draft versions of this paper. We also thank the referees for their extremely insightful comments.

Leavens's work was funded in part by the US National Science Foundation under grant CCF-0429567. Müller's work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[3] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, March 1977.

[4] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[5] Krzystof R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[7] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys'06*, pages 73–85, 2006.

[8] Fabian Y. Bannwart and Peter Müller. A logic for bytecode. In Fausto Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141(1) of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[9] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. FMCO 2005, to appear. Available from `http://research.microsoft.com/~leino/papers/krml160.pdf`, 2006.

[10] 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, 2004.

[11] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *Program Analysis For Software Tools and Engineering (PASTE)*, pages 82–87. ACM, September 2005.

[12] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[13] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction (MPC)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, July 2004.

[14] Bernhard Beckert. A dynamic logic for Java Card. In Sophia Drossopoulou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Workshop on Formal Techniques for Java Programs (FTfJP)*. Technical Report 269, FernUniversität Hagen, 2000.

[15] Bernhard Beckert and Bettina Sasse. Handling Java's abrupt termination in a sequent calculus for Dynamic Logic. In Bernhard Beckert, Robert France, Reiner Hähnle, and Bart Jacobs, editors, *IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development*, pages 5–14, 2001.

[16] Nick Benton. A typed, compositional logic for a stack-based abstract machine. In Kwangkeun Yi, editor, *Programming Languages and Systems: Third Asian Symposium (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 364–380. Springer-Verlag, November 2005.

[17] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.

[18] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[19] Manfred Broy and Martin Wirsing. Partial abstract types. *Acta Informatica*, 18(1):47–64, November 1982.

[20] Martin Büchi. Safe language mechanisms for modularization and concurrency. Technical Report TUCS Dissertations No. 28, Turku Center for Computer Science, May 2000.

[21] Martin Büchi and Wolfgang Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999. http://www.abo.fi/~mbuechi/publications/TR297.html.

[22] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[23] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: a developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Formal Methods (FME)*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, September 2003.

[24] Cristiano Calcagno, Peter O'Hearn, and Richard Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(2):557–581, 2003.

[25] Patrice Chalin. Improving JML: For a safer and more effective language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Formal Methods (FME)*, volume 2805 of *Lecture Notes in Computer Science*, pages 440–461. Springer-Verlag, 2003.

[26] Roderick Chapman. Industrial experience with SPARK. *ACM SIGADA Ada Letters*, 20(4):64–68, 2000.

[27] Julien Charles. Adding native specifications to JML. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2006.

[28] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, April 2003. The author's Ph.D. dissertation.

[29] Yoonsik Cheon, Yoshiki Hayashi, and Gary T. Leavens. A thought on specification reflection. In N. Callaos, W. Lesso, and B. Sanchez, editors, *The 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Volume II, Computing Techniques*, pages 485–490, 2004.

[30] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, May 2005.

[31] Dave G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 37, number 11 in *SIGPLAN Notices*, pages 292–310. ACM, November 2002.

[32] Dave G. Clarke, John. M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33, number 10 in *SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[33] Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag, New York, NY, 1990.

[34] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[35] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 296–300. Springer Verlag, 2005.

[36] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426. ACM, June 2006.

[37] Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[38] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[39] 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, 2005.

[40] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[41] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[42] C# language specification. ECMA Standard 334, June 2005.

[43] Eiffel analysis, design and programming language. ECMA Standard 367, June 2005.

[44] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, NY, 1985.

[45] Michael Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[46] Loe M. G. Feijs and Hans B. M. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1992.

[47] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[48] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, March 2001.

[49] 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.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[51] Joseph A. Goguen, James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[52] Aaron Greenhouse. A programmer-oriented approach to safe concurrency. Technical Report CMU-CS-03-135, School of Computer Science, Carnegie Mellon University, May 2003.

[53] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 205–229. Springer-Verlag, June 1999.

[54] David Gries. *The Science of Programming*. Springer-Verlag, New York, NY, 1981.

[55] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1994.

[56] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[57] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[58] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. Available from `http://www.cs.utoronto.ca/~hehner/aPToP`.

[59] Eric C. R. Hehner. Specified blocks. Verified Software: Theories, Tools, Experiments (VSTTE), `http://vstte.inf.ethz.ch/Files/hehner.pdf`, October 2005.

[60] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[61] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[62] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, January 2003.

[63] Tony Hoare, Jayadev Misra, and N. Shankar. Verified software: Theories, tools, experiments (VSTTE 2005). `http://vstte.ethz.ch`, October 2005. Sponsored by International Federation for Information Processing, Technical Committee 2.

[64] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[65] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE)*, pages 284–303. Springer-Verlag, 2000.

[66] Bart Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[67] Bart Jacobs, Joseph Kiniry, and M. Warnier. Java program verification challenges. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO 2002: Formal Methods for Component Objects, Proceedings*, volume 2852 of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, 2003.

[68] Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: Proof rules and implementation. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2005.

[69] Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2006.

[70] Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.

[71] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[72] Kevin D. Jones. LM3: A Larch interface language for Modula-3: A definition and introduction: Version 1.0. Technical Report 72, Digital Equipment Corporation, Systems Research Center, June 1991.

[73] Miguel Katrib and Jesús Coira. Improving Eiffel assertions using quantified iterators. *Journal of Object-Oriented Programming*, 10(7):35–43, November 1997.

[74] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2005.

[75] Reto Kramer. iContract – the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE Computer Society Press, August 1998.

[76] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. Technical Report 06-22, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, August 2006. To appear in the proceedings of *ICFEM'06*.

[77] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[78] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science, January 2006. To appear in *ACM SIGSOFT Software Engineering Notes*.

[79] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–208, March 2005.

[80] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[81] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, August 2006.

[82] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, January 2006.

[83] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[84] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[85] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In Benjamin Pierce, editor, *Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1997. Available from: `http://www.cis.upenn.edu/~bcpierce/FOOL/`.

[86] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[87] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, June 2004.

[88] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, July 2005.

[89] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, March 2006.

[90] 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.

[91] 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.

[92] 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 (FTfJP)*, Technical Report 251. FernUniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.

[93] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[94] David Luckham. *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1990.

[95] David Luckham and Friedrich W. von Henke. An overview of Anna — a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[96] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/-JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, January–March 2004.

[97] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[98] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[99] Marcello Miragliotta. Specification model library for the interactive program prover JIVE. Student project, ETH Zurich. Available from: `http://www.sct.inf.ethz.ch/projects/student_docs/Marcello_Miragliotta/Marcello_Miragliotta_paper.pdf`, 2004.

[100] Carroll Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[101] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[102] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency & Computation: Practice & Experience*, 15(2):117–154, February 2003.

[103] Peter Müller, Arnd Poetzsch-Heffter, and Gary 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.

[104] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*. Elsevier, July 2001.

[105] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 268–280. ACM, January 2004.

[106] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.

[107] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2002.

[108] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–258. ACM, January 2005.

[109] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. Doaitse Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[110] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 115–128. ACM, June 2003.

[111] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[112] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[113] Steve Shaner. Semantics for model programs in JML. Master's thesis, Iowa State University, 2006. Expected.

[114] Neelam Soundarajan and Stephen Fridella. Incremental reasoning for object oriented systems. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 302–333. Springer-Verlag, 2004.

[115] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992.

[116] Tim Wahls, Albert L. Baker, and Gary T. Leavens. The direct execution of SPECS-C++: A model-based specification language for C++ classes. Technical Report 94-02b, Department of Computer Science, Iowa State University, March 1994.

[117] Mitchell Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19(1):27–44, August 1979.

[118] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[119] Thomas Wilson, Savi Maharaj, and Robert G. Clark. Omnibus verification policies: A flexible, configurable approach to assertion-based software verification. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 150–159. IEEE Computer Society, September 2005.

[120] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[121] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[122] Yichen Xie and Alexander Aiken. Scalable error detection using boolean satisfiability. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 351–363. ACM, January 2005.