How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification

Gary T. Leavens, Yoonsik Cheon, Curtis Clifton,
Clyde Ruby, and David R. Cok

TR #03-04a
March 2003, revised March 2004

Keywords: specification languages, runtime assertion checking, documentation, tools, formal methods, program verification, programming by contract, Java language, JML language, Eiffel language, Larch family of specification languages.

2001 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — modules and interfaces, object-oriented design methods; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, validation, JML; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, mechanical verification, pre- and post-conditions, specification techniques.

This is a slightly corrected version of the paper that appears in Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (eds.), *Formal Methods for Components and Objects*, pages 262-284. Volume 2852 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, 2003. See `http://www.springer.de/comp/lncs/index.html`.

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

How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification

Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby,[1] and David R. Cok[2]

[1] Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, Iowa 50011-1041 USA
{leavens, cheon, cclifton, ruby}@cs.iastate.edu
phone: +1 515 294 1580, fax: +1 515 294 1580
[2] Eastman Kodak Company
Research & Development Laboratories
1700 Dewey Avenue, Building 65, Rochester, New York 14650-1816 USA
david.cok@kodak.com
phone: +1 585 588 3107, fax: +1 585 588 3269

Abstract. Specifications that are used in detailed design and in the documentation of existing code are primarily written and read by programmers. However, most formal specification languages either make heavy use of symbolic mathematical operators, which discourages use by programmers, or limit assertions to expressions of the underlying programming language, which makes it difficult to write complete specifications. Moreover, using assertions that are expressions in the underlying programming language can cause problems both in runtime assertion checking and in formal verification, because such expressions can potentially contain side effects. The Java Modeling Language, JML, avoids these problems. It uses a side-effect free subset of Java's expressions to which are added a few mathematical operators (such as the quantifiers `\forall` and `\exists`). JML also hides mathematical abstractions, such as sets and sequences, within a library of Java classes. The goal is to allow JML to serve as a common notation for both formal verification and runtime assertion checking; this gives users the benefit of several tools without the cost of changing notations.

1 Introduction

The Java Modeling Language, JML [55,54], is the result of a cooperative, international effort aimed at providing a common notation and semantics for the specification of Java code at the detailed-design level [58]. JML is being designed cooperatively so that many different tools can use a common notation for Hoare-style behavioral interface specifications. In this paper we explain the features of JML's design that make its assertions easily understandable by programmers and suitable for both runtime assertion checking and formal verification.

1.1 Background

By a *Hoare-style* specification we mean one that uses pre- and postconditions to specify the behavior of methods [34,43,44]. A *behavioral interface specification language* (BISL) is a specification language that specifies both the syntactic interface of a module and its behavior [33,48,52,85]. JML, the interface specification languages in the Larch family [33,48,52,85] and RESOLVE/C++ [22,73] are BISLs. Most design by contract languages and tools, such as Eiffel [70,71] and APP [77], are also BISLs, because they place specifications inside programming language code. By contrast, neither Z [80,79,87] nor VDM [6,27,74,43] is a BISL; they have no way to specify interface details for a particular programming language. OCL [82,83] is a BISL for the UML, but the UML itself is language-independent; this poses problems for a Java programmer, because the UML does not have standard notations for all details of Java method signatures. For example, the UML's syntax for specifying the signatures of operations has no standard notation for declaring that a Java method is `strictfp` or for declaring the exceptions that a method may throw [7, pp. 128-129] [49, p. 516].[1] Also the OCL has no standard constraints that correspond to JML's exceptional postconditions. Because BISLs like JML specify both interface and behavior, they are good at specifying detailed designs that include such Java details. This makes JML well suited to the task of documenting reusable components, libraries, and frameworks written in Java.

1.2 Tool Support

Because BISLs are easily integrated with code, they lend themselves to tool support for activities related to detailed design, coding, testing, and maintenance. An important goal of JML is to enable a wide spectrum of such tools. Besides tools that enforce JML's semantics (e.g., type checking), the most important JML tools help with the following tasks.

Runtime checking and testing. The Iowa State group provides (from `www.jmlspecs.org`):

- the jmlc runtime assertion checking compiler [13], which generates class files from JML-annotated Java sources,[2] and
- the jmlunit tool [14], which uses the runtime assertion checker to generate test oracle code for JUnit tests.

Documentation. David Cok provides the jmldoc tool, also available through `www.jmlspecs.org`, which generates HTML documentation similar to that

[1] Larman notes that the UML has some nonstandard ways to specify the exceptions that a method may throw, by either using Java's own syntax directly or by using a "property string".

[2] Besides this runtime assertion checking work at Iowa State, which relies on adding instrumentation to compiled code, Steven Edwards's group at Virginia Tech is working on a wrapper-class based approach to runtime assertion checking that will allow instrumentation of programs for which source code is not available.

produced by javadoc [29], but containing specifications as well. The generated documentation is useful for browsing specifications or posting to the web.

Static analysis and verification. The following tools are prepared by our partners at Compaq and the University of Nijmegen:

- The ESC/Java tool [28,65,66] statically checks Java code for likely errors. ESC/Java understands a subset of JML annotations.
- The LOOP tool [37,38,40,42] assists in the formal verification of the correctness of implementations from JML specifications, using the theorem prover PVS.

In addition, the Daikon dynamic invariant detector [23,72] outputs invariants for Java programs in a subset of JML, and the Korat automated testing tool [8] uses the jmlunit tool to exercise the test data it derives.

In this paper, we discuss how JML meets the needs of tools for runtime assertion checking, documentation, static analysis, and verification. We focus on runtime assertion checking and formal verification, which we consider to be the extremes of the spectrum of tools that a BISL might support. The tasks of runtime assertion checking and formal verification have widely differing needs:

- Runtime assertion checking places a high premium on executability. Many specification languages intended for runtime assertion checking, such as Eiffel [70,71] and APP [77], only allow assertions that are completely executable. This is sensible for a language that is intended only to support runtime assertion checking and not formal verification.
- On the other hand, formal theorem proving and reasoning place a high premium on the use of standard mathematical notations. Thus, most specification languages intended for formal reasoning or verification, such as VDM, the members of the Larch family, and especially Z, feature a variety of symbolic mathematical notations. Many expressive mathematical notations, such as quantifiers, are impossible, in general, to execute at runtime. Again, including such notations is sensible for a language intended only to support formal theorem proving and reasoning and not runtime assertion checking.

1.3 Problems

We begin by describing the problems that arise when addressing the needs of the range of tools exemplified by runtime assertion checking and formal verification.

1.3.1 Notational Problem It is often said that syntax does not matter; however, our experience with Larch/Smalltalk [11] and Larch/C++ [12,50,51,53,56] showed that programmers object to learning a specialized mathematical notation (the Larch Shared Language). This is similar to the problems found by Finney [26], who did a preliminary experiment demonstrating that the symbolic notation in Z specifications may make them hard to read. Conversely, in executable languages like Eiffel and APP, programmers feel comfortable with the use of the

programming language's expressions in assertions. Such an assertion language is therefore more appealing for purposes of documentation than highly symbolic mathematical notations.

To summarize, the first problem that we address in this paper is how to provide a good syntax for specification expressions. *Specification expressions* are the syntactic forms that are used to denote values in assertions. By a *good* syntax we mean one that is close enough to programming language expressions that programmers feel comfortable with it and yet has all of the features necessary to support both runtime assertion checking and formal verification.

1.3.2 Undefinedness Problem Expressions in a programming language may abruptly terminate (e.g., throw exceptions) and may go into infinite loops; consequently, they may have undefined values from a strictly mathematical point of view. Programming languages typically provide features to deal explicitly with such undefinedness. For example, Java provides short-circuit versions of boolean operators (such as `&&` and `||`) that allow programmers to suppress evaluation of some subexpressions.

We want both programmers and mathematicians to use JML's notations; hence, JML's specification expressions should not only look like Java's expressions and use Java's semantics, but should also validate the standard laws of logic. However, because of a potential for undefinedness, Java expressions do not satisfy all the standard rules of logic; for example, in Java the conjunction E_1 `&&` E_2 is not equal to E_2 `&&` E_1, although in logic they would be equal. To resolve this conflict, we are willing to accept a slightly different semantics for assertion evaluation as long as programmers are not too surprised by it.

Thus, the second problem we address in this paper is how to find a semantics for expressions used in assertions that validates standard laws of logic and yet does not surprise programmers and is still useful for runtime assertion checking.

1.3.3 Side Effects Problem Another important semantic issue is that expressions in a programming language like Java (and most others, including Eiffel) can contain side effects. Side effects have a very practical problem related to runtime assertion checking. It is generally assumed that assertions may be evaluated or skipped with no change in the outcome of a computation, but an assertion with side effects has the potential to alter the computation's outcome. For example, an assertion with side effects might mask the presence of a bug that would otherwise be revealed or cause bugs that are not otherwise present. Because one of the principal uses of runtime assertion checking is debugging and isolating bugs, it is unacceptable for side effects from assertion checking to alter the outcome of a computation.

Thus, the third problem that we address in this paper is how to prevent side effects in assertions while still retaining as much of the syntax of normal programming language expressions as possible.

1.3.4 Mathematical Library Problem

Most specification languages come with a library of mathematical concepts such as sets and sequences. Such concepts are especially helpful in specifying collection types. For example, to specify a `Stack` type, one would use a mathematical sequence to describe, abstractly, the states that a stack object may take [35]. VDM, OCL, Z, and the interface specification languages of the Larch family all have libraries of such mathematical concepts. They also are standard in theorem provers such as PVS.

However, as discussed in Section 1.3.1, we want to limit the barriers that Java programmers must overcome to use JML. Thus, the fourth problem that we address in this paper is how to provide a library of mathematical concepts in a way that does not overwhelm programmers, and yet is useful for formal verification.

1.4 Other Goals of JML

In addition to providing solutions to the preceding four problems, the design of JML is guided and constrained by several other goals. One of the most important of these goals is to allow users to write specifications that document detailed designs of existing code. This motivates the choice of making JML a BISL, as described above. Moreover, we would like JML to be useful for documenting code regardless of whether it was designed according to any particular design method or discipline. This is important because the cost of specification is high enough that it is not always justified until one knows that the design and the code have stabilized enough to make the documentation potentially useful to other people.

In general, JML's design adheres to the goal of being able to document existing designs; however, there is one significant aspect of JML's design that departs from this goal—JML imposes the specifications of supertypes on subtypes, a property termed *specification inheritance*, in order to achieve behavioral subtyping [19].

JML's use of specification inheritance is justified by another of our goals: we want JML to support *modular reasoning*, that is, reasoning about the behavior of a compilation unit using just the specifications of the compilation units that it references (as opposed to the details of their implementations). Modular reasoning is important because without it, the difficulty of understanding an object-oriented program increases much more rapidly than the size of the program, and thus the benefits of the abstraction mechanisms in object-oriented languages are lost. Consequently, modular reasoning is also important for formal verification, because then the scope of the verification problem is limited.

Specification inheritance, and the resulting behavioral subtyping, allows modular reasoning to be sound, by allowing one to reason based on the static types of references. Subsumption in Java allows a reference to a subtype object to be substituted for a supertype reference. The requirements of behavioral subtyping [2,3,19,57,59,63,69] guarantee that all such substituted objects will obey the specifications inherited from the static type of the reference [19,60,61].

Because of the benefits of modular reasoning to programmers and verifiers, we favor specification inheritance over the conflicting goal of being able to document

existing designs that do not follow behavioral subtyping. In any case, it is possible to work around the requirements of behavioral subtyping for cases in which a subtype does not obey the inherited specifications of its supertype(s). One simply underspecifies each supertype enough to allow all of the subtypes that are desired [63,69]. Note that this work-around does not involve changing the code or the design, but only the specification, so it does not interfere with the goal of documenting existing code.

1.5 Outline

The remainder of this paper is organized as follows. The next section discusses our solution to the notational problem described above. Having described the notation in general terms, Section 3 provides more background on JML. The subsequent three sections treat the remaining problems discussed above. The paper ends with a discussion of related work and some conclusions.

2 Solving the Notational Problem

To solve the notational problem described in Section 1.3.1, JML generally follows Eiffel, basing the syntax of specification expressions on Java's expression syntax. However, because side effects are not desired in specification expressions, JML's specification expressions do not include Java expressions that can cause obvious side effects, i.e., assignment expressions and Java's increment and decrement operators (`++` and `--`).

Furthermore, to make JML suitable for formal verification efforts, JML includes a number of operators that are not present in Java [55, Section 3]. The syntax of these operators comes in two flavors: those that are symbolic and those that are textual.

We did not want to introduce excess notation that would cause difficulties for programmers when reading specifications, so JML adds just five symbolic operators. Four of these are logical operators: forward and reverse implication, written `==>` and `<==`, respectively, and logical equivalence and inequivalence, written `<==>` and `<=!=>`, respectively. The inclusion of symbols for logical operators is inspired by the calculational approach to formal methods [17,20,31]. The other symbolic operator is `<:`, which is used to compare types to see if they are in a subtype relationship [65].

All the other operators added to Java and available in JML's specification expressions use a textual notation consisting of a backslash (\) followed by an English word or phrase. For example, the logical quantifiers in JML are written as `\forall` and `\exists` [55].

Besides these quantifiers, JML also has several other operators using this backslash syntax. One of the most important is `\old()`, which is used in method postconditions to indicate an expression whose value is taken from the pre-state of a method call. For example, `\old(i-1)` denotes the value of `i-1` evaluated in the pre-state of a method call. This notation is borrowed from the `old` operator

in Eiffel. Other JML expressions using the backslash syntax include `\fresh(o)`, which says that `o` was not allocated in the pre-state of a method call, but is allocated (and not null) in the post-state, and `\result`, which denotes the normal result returned by a method.

The backslashes in the syntax of these operators serve a very important purpose—they prevent the rest of the operator's name from being interpreted as a Java identifier. This allows JML to avoid reserving Java identifiers in specification expressions. For example, `result` can be used as a program variable and is distinguished from `\result`. This trick is useful in allowing JML to specify arbitrary Java programs. Indeed, because a goal of JML is to document existing code, it cannot add new reserved words to Java.

3 Background on JML

In this section we provide additional background on JML that will be useful in understanding our solutions to the remaining problems.

3.1 Semantics Of Specification Expressions

Just as JML adopts much of Java's expression syntax, it attempts to keep JML's semantics similar to Java's. In particular, the semantics of specification expressions is a reference semantics. That is, when the name of a variable or field is used in an expression, it denotes either a primitive value (such as an integer) or a reference to an object. References themselves are values in the semantics, which allows one to directly express aliasing or the lack of it. For example, the expression `arg != fieldVal` says that `arg` and `fieldVal` are not aliased. Java also allows one to compare the states of objects using the `equals` method. For example, in the postcondition of a `clone` method, one might write the following to say that the result returned by `clone` is a newly allocated object that has the same state as the receiver (`this`):

```
\fresh(\result) && this.equals(\result);
```

Note that the exact meaning of the `equals` method for a given type is left to the designer of that type, as in Java. Thus, if one only knows that `o` is an `Object`, it is hard to conclude much about `x` from `o.equals(x)`.

Because JML uses this reference semantics, specifiers must show the same care as Java programmers when choosing between the `==` and `equals` equality tests. And like Eiffel, but unlike Larch-style interface specification languages, JML does not need "state functions" to be applied to extract the value of an expression from a reference. Values are implicitly extracted as needed by methods and operators. Besides being easier for programmers, this lends some succinctness to the notation.

Currently, JML adopts all of the Java semantics for integer arithmetic. Thus types such as `int` use two's complement arithmetic and are finite. Although Java programmers are, in theory, aware of the nature of integer arithmetic, it

seems that JML's adoption of Java's semantics causes some misunderstandings; for example, some published JML specifications are inconsistent because of this semantics [10]. Chalin has suggested adding new primitive value types for infinite precision arithmetic to JML; in particular, he suggests a type `\bigint` for infinite precision integers [9,10]. He is currently implementing and experimenting with this idea.

3.2 Method and Type Specifications

To explain JML's semantics for method specifications, we use the example in Figure 1. JML uses special comments, called *annotations*, to hold the specification of behavior; these are added to the interface information contained in the Java code. A specifier writes these annotation comments by inserting an at-sign (`@`) following the usual characters that signify the start of a comment. In multi-line annotation comments, at-signs at the beginnings of lines are ignored.

Figure 1 starts with a "model import" directive, which says that JML will consider all types in the named package, `org.jmlspecs.models`, to be imported for purposes of the specification. This allows the JML tools to find the type `JMLObjectSequence` (see the third line) in that package.

The type `JMLObjectSequence` is used as the type of the model instance field, named `absVal`. In this declaration, the `model` keyword says that the field is not part of the Java code, but is used solely for purposes of specification. The `instance` keyword says that the field is imagined, for purposes of specification, to be a non-static field in every class that implements this interface.[3]

Following the declaration of the two model instance fields is an invariant. It says that the field `absVal` is never null.

Following the invariant are the declarations and specifications of three methods. In JML, a method's specifications are typically written, as they are in Figure 1, before the header of the method they specify. This makes the scope of the formal parameters of a method a bit strange, because it extends backward into the method's specification. However, it works best with Java tools, which expect comments related to a method, such as javadoc comments, to precede the method's header.

Consider the specification of the first method, `push`. This shows the general form of a "normal behavior" specification case. A *specification case* includes a precondition, indicated by the keyword `requires`, and some other specification clauses. A specification case is satisfied if, whenever the precondition is satisfied, the other clauses are also satisfied. Additionally, in a normal behavior specification case, the method must not throw an exception when the precondition is satisfied. The specification case given for `push` includes, besides the requires clause, a frame axiom, introduced by the keyword `assignable`, and a normal postcondition, following the keyword `ensures`.

[3] Omitting `instance` makes fields static and final, which is Java's default for fields declared in interfaces.

```
//@ model import org.jmlspecs.models.*;
public interface Stack {
  //@ public model instance JMLObjectSequence absVal;

  //@ public instance invariant absVal != null;

  /*@ public normal_behavior
    @   requires true;
    @   assignable absVal;
    @   ensures absVal.equals(\old(absVal.insertFront(x))); @*/
  void push(Object x);

  /*@   public normal_behavior
    @     requires !absVal.isEmpty();
    @     assignable absVal;
    @     ensures absVal.equals(\old(absVal.trailer()))
    @          && \result == \old(absVal.first());
    @ also
    @   public exceptional_behavior
    @     requires absVal.isEmpty();
    @     assignable \nothing;
    @     signals (Exception e)
    @          e instanceof IllegalStateException; @*/
  Object pop();

  //@ ensures \result <==> absVal.isEmpty();
  /*@ pure @*/ boolean isEmpty();
}
```

Fig. 1. The specification and code for the interface `Stack`.

As with specification languages in the Larch family, a precondition that is just true can be omitted. In the Larch family, an omitted frame axiom means "`assignable \nothing;`", which is a very strong specification that says that the method has no side effects. Following a suggestion of Erik Poll, we decided that such a specification was too strong for a default. So in JML, an omitted frame axiom allows assignment to all locations. This agrees with most of the defaults for omitted clauses in JML, which impose no restrictions.

JML also allows specifiers to write "exceptional behavior" specification cases, which say that, when the precondition is satisfied, the method must not return normally but must instead throw an exception. An example appears in the specification of the `pop` method. This specification has two specification cases connected with `also`. The meaning of the `also` is that the method must satisfy both of these specification cases [84,86]. Thus, when the value of the model instance field `absVal` is not empty, a call to `pop` must return normally and must satisfy

the given ensures clause. But when the value of the model instance field `absVal` is empty, a call to `pop` must throw an `IllegalStateException`. This kind of case analysis can be desugared into a single specification case, which can be given a semantics in the usual way [38,41,53,76].

The specification cases given for `push` and `pop` are heavyweight specification cases [55, Section 1]. Such specification cases are useful when one wants to give a relatively complete specification, especially for purposes of formal verification. For runtime assertion checking or documentation, one may want to specify only part of the behavior of a method. This can be done using JML's lightweight specification cases, which are indicated by the absence of a behavior keyword (like `normal_behavior`). Figure 1 gives an example of a lightweight specification case in the specification of the method `isEmpty`.

4 Dealing with Undefinedness

As discussed in Section 1.3.2, a fundamental problem in using the underlying language for specification expressions is dealing with expressions whose value is undefined. In Java, undefinedness in expressions is typically signaled by the expression throwing an exception. For example, when one divides an integer by 0, the expression throws an `ArithmeticException`. Exceptions may also be thrown by methods called from within specification expressions.

Specification languages have adopted several different approaches to dealing with undefinedness in expressions [4,32]. We wanted a semantics that would not be surprising to either Java programmers or to those doing formal verification. Typically, a Java programmer would try to write the specification in a way that "protects" the meaning of the expression against any source of undefinedness [62]. This can be accomplished by using the short-circuit boolean operators; for example, a specifier might write `denom > 0 && num/denom > 1` to be sure that the division would be defined whenever it was carried out.

However, we would like specifications to be meaningful even if they are not protective. Hence, the semantics of JML does not rely on the programmer writing protective specifications but, instead, ensures that every expression has some value. To do this, we adopted the "underspecified total functions" approach favored in the calculational style of formal methods [31,32]. That is, an expression that would not have a value in Java is given an arbitrary, but unspecified, value. For example, `num/0` has some integer value, although this approach does not say what the value is, only that it must be uniformly substituted in any surrounding expression. In JML all expressions have an implicit argument of the program's state; thus, the uniform substitution of values need only be carried out within a given assertion.

An advantage of this substitution approach is that it validates the rules for standard logic. For example, in JML, E_1 `&&` E_2 is equivalent to E_2 `&&` E_1. Consider what happens if E_1 throws an exception; in that case, one may chose some unspecified boolean value for E_1, say b. This means that E_1 `&&` E_2 equals b `&&` E_2, which is equal to E_2 `&&` b, as can be seen by a simple case analysis on

E_2's value. The case where E_2 throws an exception is similar. Furthermore, if programmers write protective specifications, they will never be surprised by the details of this semantics.

The JML assertion checking compiler takes advantage of the semantics of undefinedness to attempt, as much as possible, to detect possible assertion violations [13]. That is, assertion checking attempts to use a value that will make the overall assertion false, whenever the undefinedness of some subexpression allows it to do so. In this way, the assertion checker can both follow the rules of standard logic and detect places where specifications are not sufficiently protective. This is a good example of how JML caters to the needs of both runtime assertion checking and formal verification.

5 Preventing Side Effects in Assertions

As discussed in Section 1.3.3, it is important to prevent side effects in assertions, for both practical and theoretical reasons. JML is designed to prevent such side effects statically. It does this using an effect-checking type system [30,81]. This type system is designed to be as simple as possible. Although it allows specification expressions to call Java methods and constructors, it only allows such calls if the called method or constructor is declared with the modifier `pure`. The semantics of JML must thus assure that pure methods and constructors are side-effect free.

5.1 JML's Purity Restrictions

JML's semantic restrictions on pure methods and constructors are as follows:

- A *pure method* implicitly has a specification that includes the following specification case [55, Section 2.3.1]:

  ```
  assignable \nothing;
  ```

  This ensures that a correct implementation of the method has no side effects.
- "A *pure constructor* implicitly has a specification that only allows it to assign to the instance fields of the class in which it appears" (including inherited instance fields) [55, Section 2.3.1]. This ensures that, if the constructor is correctly implemented, then a `new` expression that calls it has no side effects.
- Pure methods and pure constructors may only invoke other methods and constructors that are pure. This makes the type system modular, as it allows the purity of a method or a constructor to be checked based only on its code and the specifications of the other methods and constructors that it calls.
- All methods and constructors that override a pure method or constructor must also be pure. This inheritance of purity is a consequence of specification inheritance and is necessary to make the type system modular in the presence of subtyping.

The first restriction implies that a pure method may not perform any input or output, nor may it assign to any non-local variables. Similarly, by the second restriction, a pure constructor may not do any I/O and may not assign to non-local storage other than the instance fields of the object the constructor is initializing.

Note that, in JML, saying that a method may not assign to non-local storage means precisely that—even benevolent side effects are prohibited [55, Section 2.1.3.1]. This seems necessary for sound modular reasoning [64]. It is also a useful restriction for reasoning about supertypes from their specifications [78] and for reasoning about concurrent programs.

The last two restrictions are also motivated by modularity considerations. Inheritance of purity has as a consequence that a method cannot be pure if any overriding method has side effects. In particular, a method in `Object` can be specified as pure only if every override of that method, in any Java class, obeys JML's purity restrictions.

The type system of JML is an important advance over languages like Eiffel, which trust programmers to avoid side effects in assertions rather than statically checking this property. However, as we will see in the following subsection, JML's purity restrictions give rise to some practical problems.

5.2 Practical Problems with JML's Purity Restrictions

An initial practical problem is how to decide which methods in Java's libraries should be specified as pure. One way to start to answer this question is to use a static analysis to conservatively estimate which methods in Java's libraries have side effects. A conservative analysis could count a method as having side effects if it assigns to non-local storage or calls native methods (which may do I/O), either directly or indirectly. All other methods can safely be specified as pure, provided they are not overridden by methods that the analysis says have side effects. Researchers from Purdue have provided a list of such methods to us, using their tools from the Open Virtual Machine project.[4] We hope to integrate this technology into the JML tools eventually.

Declaring a method to be `pure` entails a very strong specification, namely that the method and all possible overriding methods have no side effects. Thus, finding that a method, and all known methods that override it, obey JML's purity restrictions is not the same as deciding that the method *should* be specified as pure. Such a decision affects not just all existing overrides of the method, but all future implementations and overrides. How is one to make such a decision?

This problem is particularly vexing because there are many methods that seem intuitively to be side-effect free, but that do not obey JML's purity restrictions. Methods with benevolent side effects are common examples. A *benevolent side effect* is a change in the internal state of an object in a way that is not externally visible. Two examples from the protocol of `Object` will illustrate the importance of this problem.

[4] See `http://www.ovmj.org/`.

First, consider computing a hash code for an instance of a class. Because this may be computationally costly, an implementation may desire to compute the hash code the first time it is asked for and then cache the result in a private field of the object. When the hash code is requested on subsequent occasions, the cached result is returned without further computation. For example, this is done in the `hashCode` method of Java's `String` class. However, in JML, storing the computed hash code into the cache is considered to be a side effect. So `String`'s `hashCode` method cannot be specified as pure.

Second, consider computing object equality. In some implementations, an object's fields might be lazily initialized or computed only on first access. If the `equals` method happens to be the first such method to be called on such an object, it will trigger the delayed computation. We found such an example in our work on the MultiJava compiler [15,16]; in this compiler, the class `CClassType` has such delayed computations, and its override of `Object`'s `equals` method can trigger a previously delayed computation with side effects. It seems very difficult to rewrite this method to be side-effect free, because to do so one would probably need to change the compiler's architecture. (Similar kinds of lazy initialization of fields occur in implementations of the Singleton pattern, although these usually do not affect the `equals` method.)

We have shown two cases where methods in the protocol of `Object` are overridden by methods that cannot be pure. By purity and specification inheritance, these examples imply that neither `hashCode` nor `equals` can be specified as pure in `Object`. `Object` is typically used in Java as the type of the elements in a collection. Hence, in the specification of a collection type, such as a hash table, one cannot use the `hashCode` or `equals` methods on elements. Without changes, this would make JML unsuitable for specifying collection types.

(This problem is mostly a problem for collection types, because one can specify many subclasses of `Object` with pure `hashCode` and `equals` methods. Specifications operating on instances of such subclasses can use these methods without violating JML's type system.)

5.3 Solving the Problems

The desire to use intuitively side-effect free methods in specifications, even if they are not pure according to JML's semantics, is strong enough that we considered changing the semantics of the assignable clause in order to allow benevolent side effects. However, we do not know how to do that and still retain sound modular reasoning [64]. In any case, the use of such methods in runtime assertion checking would still be problematic because of the side effects they might cause. In addition, we would like to prevent problems when a programmer wrongly believes that side effects are benevolent; it is not clear whether an automatic static analysis could prevent such problems, and even if so, whether such a tool could be modular.

Thus far, the only viable solution we have identified is to refactor specifications by adding pure *model* (i.e., specification-only) methods that are to be used in specifications in place of program methods that cannot be pure. That

is, whenever one has an intuitively side-effect free program method, m, that is not pure according to JML's semantics, one should create a pure model method m', which returns the same result as m but without its side effects. Then one replaces calls to m by calls to m' in assertions.

We are currently experimenting with this solution. The most important part of this experiment is to replace uses of `Object`'s `equals` method, which cannot be pure, with calls to a new pure model method in `Object`, called `isEqualTo`. The specifications of these methods are shown in Figure 2. The `assignable` clause in the specification of the `equals` method permits benevolent side effects; it is also specified to return the same result as would a call to `isEqualTo`. Thus, whenever someone overrides `equals`, they should also override the `isEqualTo` method. When an override of `equals` is specified as pure, then an override of `isEqualTo` in the same class can be specified in terms of this pure `equals` method, and the implementation of the model `isEqualTo` method can simply call `equals` as well. However, an implementation of `equals` can never call `isEqualTo`, because program code cannot call model methods (since model methods can only be used in specifications). Therefore, to avoid code duplication when `equals` is not declared to be pure but the two methods share some common implementation code, one can introduce a (non-model) pure, private method that both `equals` and `isEqualTo` can call.

We have also applied this refactoring to all the collection classes in `java.util` (and in other packages) that we had previously specified, in order to check that the solution is viable. So far the results seem satisfactory. However, as of May 2003, this restructuring is not part of the JML release, because the JML tools are not yet able to handle some of the details of this approach. In particular, the runtime assertion checker is not yet able to compile the model methods added to `Object` without having all of `Object`'s source code available. (And we cannot legally ship Sun's source code for `Object` in the JML release.) However, we working on solutions to this problem that will allow us to obtain more experience with this approach and to do more case studies.

5.4 Future Work on Synchronized Methods and Purity

JML currently permits synchronized methods to be declared `pure` if they meet all the criteria described in Section 5.1. Given that obtaining a lock is a side effect that can affect control flow in a program, does allowing synchronized methods to be pure violate the intent of JML's purity restrictions? On the surface it would seem so, because when a synchronized method gains a lock, it may change the outcome of other concurrent threads. Furthermore, execution of such a method might block, conceivably even causing a deadlock between concurrent threads that would not occur if one was not doing assertion checking. However, since we have largely ignored concurrency thus far in JML's design, we leave resolution of this issue for future work.

```
/*@   public normal_behavior
  @     assignable objectState;
  @     ensures \result <==> this.isEqualTo(obj);
  @*/
public boolean equals(Object obj);

/*@   public normal_behavior
  @       requires obj != null;
  @       assignable \nothing;
  @       ensures (* \result is true when obj is equal to this object *);
  @  also
  @    public normal_behavior
  @       requires obj != null && \typeof(this) == \type(Object);
  @       assignable \nothing;
  @       ensures \result <==> this == obj;
  @  also
  @    public normal_behavior
  @       requires obj == null;
  @       assignable \nothing;
  @       ensures \result <==> false;
  public pure model boolean isEqualTo(Object obj) {
     return this == obj;
  }
  @*/
```

Fig. 2. The refactored specification for `Object's equals` method and the pure model method `isEqualTo`. The text between `(*` and `*)` in the first specification case of `isEqualTo`'s specification is an "informal description", which formally is equivalent to writing `true` [53].

6 Mathematical Libraries

As described in Section 1.3.4, we need to provide a library of mathematical concepts with JML in a way that does not overwhelm programmers, and yet is useful for formal verification.

6.1 Hiding the Mathematics

It is sometimes convenient to use mathematical concepts such as sets and sequences in specification, particularly for collection classes [36,68,85]. For example, the specification of `Stack` in Figure 1 uses the type `JMLObjectSequence`, which is part of JML's `org.jmlspecs.models` package. This package contains types that are intended for such mathematical modeling. Besides sequences, these include sets, bags, relations, and maps, and a few other convenience types.

Most types in `org.jmlspecs.models` have only pure methods and constructors.[5] For example, `JMLObjectSequence`'s `insertFront` method returns a sequence object that is like the receiver, but with its argument placed at the front; the receiver is not changed in any way. `JMLObjectSequence`'s `trailer` method similarly returns a sequence containing all but the first element of the receiver, without changing the receiver. Because such methods are pure, they can be used during runtime assertion checking without changing the underlying computation.

JML gains two advantages from having these mathematical modeling types in a Java package, as opposed to having them be purely mathematical concepts. First, these types all have Java implementations and thus can be used during runtime assertion checking. Second, using these types in assertions avoids the introduction of special mathematical notation; instead, normal Java expressions (method calls) are used to do things like concatenating sequences or intersecting sets. This is an advantage for our main audience, which consists of programmers and not mathematicians.

6.2 Use by Theorem Provers

The second part of the mathematical libraries problem described in Section 1.3.4 is that the library of mathematical modeling types should be useful for formal verification. The types in the `org.jmlspecs.models` package are intended to correspond (loosely) to the libraries of mathematical concepts found in theorem provers, such as PVS. As we gain experience, we can add additional methods to these types to improve their correspondence to these mathematical concepts. It is also possible to add new packages of such types tailored to specific theorem provers or to other notations, such as OCL.

When translating specification expressions into theorem prover input, the Loop tool currently treats all methods in the same way — it does not make a special case for pure methods in the `org.jmlspecs.models` package. This makes the resulting proof obligations more complex than is desirable. Since the types in the models package are known, it seems that one should be able, as a special case, to replace the general semantics of such a method call with a call to some specific function from the theorem prover's library of mathematical concepts. To facilitate this, it may be that these model types should all be declared to be final, which is currently not the case.

7 Related Work

We have already discussed how JML differs from conventional formal specification languages, such as Z [80,79,87], VDM [6,27,74,43], the Larch family [33,48,52,85] and RESOLVE [22,73]. To summarize, the main difference is that

[5] The `org.jmlspecs.models` package does have some types that have non-pure methods. These are various kinds of iterators and enumerators. The methods of these iterators and enumerators that have side effects cannot be used in specification expressions.

JML's specification expressions are based on a subset of the Java programming language, a design that is more congenial to Java programmers.

The Alloy Annotation Language (AAL) offers a syntax similar to JML for annotating Java programs [46]. AAL supports extensive compile-time checking based on static analysis techniques. Unlike similar static analysis tools such as ESC/Java [18], AAL also supports method calls and relational expressions in assertions. However, AAL's assertion language is based on a simple first-order logic with relational operators [39] and not on a subset of Java expressions. We believe that a Java-based syntax is more likely to gain acceptance among Java programmers. However, JML could adopt some of AAL's features for specifying sets of objects using regular expressions. These would be helpful in using JML's frame axioms, where they would allow JML to more precisely describe locations that can be assigned to in the method. (Another option that would have similar benefits would be to use the approach taken in DemeterJ [67].)

We have also discussed how JML differs from design by contract languages, such as Eiffel [70,71], and tools, such as APP [77]. Summarizing, JML provides better support for complete specifications and formal verification by

- extending the set of specification expressions with more expressive mathematical constructs, such as quantifiers,
- ensuring that specification expressions do not contain side effects, and
- providing a library of types corresponding to mathematical concepts.

JML's specification-only (model) declarations and frame axioms also contribute to its ability to specify types more completely than is easily done with design by contract tools.

We know of several other design by contract tools for Java [5,21,24,45,47,75]. The approaches vary from a simple assertion mechanism similar to the `assert` macros of C and C++ to full-fledged contract enforcement capabilities. Jass [5], iContract [47], and JContract [75] focus on the practical use of design by contract in Java. Handshake and jContractor focus on implementation techniques such as library-based on-the-fly instrumentation of contracts [21,45]. Contract Java focuses on properly blaming contract violations [24,25]. These notations and tools suffer from the same problems as Eiffel. That is, none of them guarantee the lack of side effects in assertions, handle undefinedness in a way that would facilitate formal verification and reasoning, support more expressive mathematical notations such as quantifiers, or provide a set of immutable types designed for use in specifications. In sum, they all focus on runtime checking, and thus it is difficult to write complete specifications for formal verification and reasoning.

8 Conclusion

JML synthesizes the best from the worlds of design by contract and more mathematical specification languages. Because of its expressive mathematical notations, its specification-only (model) declarations, and library of mathematical modeling types, one can more easily write complete specifications in JML than

in a design by contract language, such as Eiffel. These more complete specifications, along with JML's purity checking, allow JML to be useful for formal verification. Thus, JML's synthesis of features allows it to serve many roles in the Java formal methods community.

Our experience so far is that this approach has had a modest impact. Release 3.7 of JML has been downloaded almost 400 times. JML has been used in at least 5 universities for teaching some aspects of formal methods. It is used somewhat extensively in the Java Smart Card industry and has been used in at least one company outside of that industry (Fulcrum).

In the future, we would like to extend the range of tools that JML supports to include tools for model checking and specification of concurrent Java programs [1]. We invite others to join us in this effort to furnish Java programmers with a single notation that can be used by many tools.

Acknowledgments

The work of Leavens, Cheon, Clifton, and Ruby was supported in part by the US National Science Foundation, under grants CCR-0097907 and CCR-0113181.

Thanks to Robyn Lutz, Sharon Ryan, and Janet Leavens for comments on earlier drafts of this paper. Thanks to all who have contributed to the design and implementation of JML including Al Baker, Erik Poll, Bart Jacobs, Joe Kiniry, Rustan Leino, Raymie Stata, Michael Ernst, Gary Daugherty, Arnd Poetzsch-Heffter, Peter Müller, and others acknowledged in [55].

References

1. E. Abraham-Mumm, F.S. de Boer, W.P. de Roever, , and M. Steffen. A tool-supported proof system for multithreaded java. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO 2002: Formal Methods for Component Objects, Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
2. Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, volume 276.
3. Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. 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, New York, NY, 1991.
4. H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, October 1984.
5. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001. Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001.

6. Juan Bicarregui, John S. Fitgerald, Peter A. Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer-Verlag, New York, NY, 1994.
7. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley Longman, Reading, Mass., 1999.
8. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM, July 2002.
9. Patrice Chalin. Back to basics: Language support and semantics of basic infinite integer types in JML and Larch. Technical Report CU-CS 2002-003.1, Computer Science Department, Concordia University, October 2002.
10. Patrice Chalin. Improving JML: For a safer and more effective language. Technical Report 2003-001.1, Computer Science Department, Concordia University, March 2003.
11. Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.
12. Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.
13. Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.
14. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.
15. Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. The author's masters thesis.
16. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, NY, October 2000. ACM.
17. Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag, New York, NY, 1990.
18. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.
19. 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.
20. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and program semantics*. Springer-Verlag, NY, 1990.

21. Andrew Duncan and Urs Holzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA, December 1998.
22. Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part II: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, Oct 1994.
23. 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.
24. Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1–15, October 2001.
25. Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), September 10-14, 2001, Vienna, Austria*, September 2001.
26. Kate Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.
27. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools in Software Development*. Cambridge, Cambridge, UK, 1998.
28. 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 ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 17–19 2002. ACM Press.
29. Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1–2 June 1995*, pages 151–173. Springer, 1995.
30. David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, pages 28–38. ACM, August 1986.
31. David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1994.
32. David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 366–373. Springer-Verlag, New York, NY, 1995.
33. John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.
34. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
35. C. A. R. Hoare. Notes on data structuring. In Ole-J. Dahl, E. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press, Inc., New York, NY, 1972.
36. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

37. Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. Ipa dissertation series, 2001-03, University of Nijmegen, Holland, February 2001.
38. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303. Springer-Verlag, 2000. An earlier version is technical report CSI-R9912.
39. Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.
40. 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, Berlin, 2003.
41. Bart Jacobs and Eric Poll. A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy, 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.
42. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 329–340. ACM, October 1998.
43. Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
44. H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, New York, NY, October 1991.
45. Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference on Reflection '99, Saint-Malo, France, July 19–21, 1999, Proceedings*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer-Verlag, July 1999.
46. Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proceedings of OOPSLA '02 Conference on Object-Oriented Programming, Languages, Systems, and Applications*, volume 37(11) of *SIGPLAN Notices*, pages 231–245, New York, NY, November 2002. ACM.
47. Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.
48. Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.
49. Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, second edition edition, 2002.
50. Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

51. Gary T. Leavens. Larch/C++ Reference Manual. Version 5.41. Available in `ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz` or on the World Wide Web at the URL `http://www.cs.iastate.edu/~leavens/larchc++.html`, April 1999.
52. Gary T. Leavens. Larch frequently asked questions. Version 1.110. Available in `http://www.cs.iastate.edu/~leavens/larch-faq.html`, May 2000.
53. Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.
54. 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, Boston, 1999.
55. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State University, Department of Computer Science, May 2003. This is an obsolete version.
56. Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch, July, 1992*, Workshops in Computing, pages 159–184. Springer-Verlag, New York, NY, 1993.
57. 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.
58. Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.
59. Gary T. Leavens and Don Pigozzi. A complete algebraic characterization of behavioral subtyping. *Acta Informatica*, 36:617–663, 2000.
60. Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.
61. 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.
62. Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10:59–75, 1998.
63. Gary Todd Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.
64. K. Rustan M. Leino. A myth in the modular specification of programs. Technical Report KRML 63, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, November 1995. Obtain from the author, at leino@microsoft.com.
65. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000.

66. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999.
67. Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, October 2001.
68. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.
69. Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
70. Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
71. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.
72. 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.
73. William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, October 1994.
74. International Standards Organization. Information technology – programming languages, their environments and system software interfaces – Vienna Development Method – specification language – part 1: Base language. ISO/IEC 13817-1, December 1996.
75. Parasoft Corporation. Using design by contract™ to automate Java™ software and component testing. Available from `http://www.parasoft.com/jsp/products/tech_papers.jsp?product=Jcontract`, as of Feb. 2003.
76. Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, August 2001. This is an obsolete version.
77. D. S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.
78. Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, October 2000.
79. J. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.
80. J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, 1989. ISBN 013983768X.
81. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.
82. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Reading, Mass., 1999.
83. Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.
84. Alan Wills. Capsules and types in Fresco: Program validation in Smalltalk. In P. America, editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, New York, NY, 1991.

85. Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.
86. Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.
87. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.