A Runtime Assertion Checker for the Java Modeling Language (JML)

Yoonsik Cheon and Gary T. Leavens

TR #02-05a
March 2002, revised April 2002

Keywords: runtime assertion checking, formal methods, formal interface specification, programming by contract, JML language, Java language

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, monitors, testing tools, theory; 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, pre- and post-conditions, specification techniques.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

A Runtime Assertion Checker for the Java Modeling Language (JML)

Yoonsik Cheon and Gary T. Leavens[*]
Department of Computer Science, Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040, USA
{cheon,leavens}@cs.iastate.edu

Abstract

Debugging is made difficult by the need to precisely describe what each piece of the software is supposed to do, and to write code to defend modules against the errors of other modules; if this is not done it is difficult to assign blame to a small part of the program when things go wrong. Similarly, unit testing also needs precise descriptions of behavior, and is made difficult by the need to write test oracles. However, debugging and testing consume a significant fraction of the cost of software development and maintenance efforts. Inadequate debugging and testing also contribute to quality problems. We describe a runtime assertion checker for the Java Modeling Language (JML) that helps in assigning blame during debugging and in automatic generation of test oracles. It represents a significant advance over the current state of the art, because it can deal with very abstract specifications which hide representation details, and other features such as quantifiers, and inheritance of specifications. Yet JML specifications have a syntax that is easily understood by programmers. Thus, JML's runtime assertion checker has the potential for decreasing the cost of debugging and testing.

Keywords: runtime assertion checking, formal interface specification, design by contract, specification inheritance, Java Modeling Language (JML)

1. Introduction

Writing formal interface specifications of program modules such as classes and interfaces can improve the quality of software designs and thus contribute to the quality of software. This process can help clarify the assumptions that a module makes about its clients and environment; it also helps one identify the module's responsibilities and obligations to its clients [2, 18, 20]. Identifying and precisely specifying responsibilities of modules often leads to a better design that is less-coupled and more cohesive. The resulting formal specification is a detailed design document that is abstract, precise, and concise; besides its value during development, such detailed design documentation is especially valuable at the maintenance phase. Some form of specification is also necessary for deciding the success or failure of tests [20].

However, formal interface specifications are seldom used by software practitioners. Although this is not the only reason, one problem is that the payoff for writing formal specifications is not immediate. So, our goal is to allow programmers to reap benefits from specifications as soon as the specifications are written. In particular, we aim to provide programmers with benefits in debugging and unit testing, costly activities that consume much of the time and effort in writing and maintaining software. When this is done, we believe that some of the other side benefits of formal specifications, in particular their value as documentation and as an aid in reasoning, will become apparent. In turn this may also help lower costs and improve software quality.

One technique that helps to produce an immediate payoff for writing formal interface specifications is to check specification assertions during the execution of programs. A formal interface specification is just a mathematical formula, but it becomes useful for testing and debugging when it can be executed to check the validity of an implementation. Checking assertions at runtime is a practical and effective means for debugging programs, as Meyer and others have emphasized [20, 22]. It also helps one debug the specifications themselves, and thus improves the quality and accuracy of documentation. Also, checking assertions at runtime can help automate parts of testing [3]. Finally, executing formal specifications is much more practical than using them for formal verification of correctness.

In this paper, we describe our experience developing a runtime assertion checker for the Java Modeling Language

[*]The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea, and by the US National Science Foundation under grants CCR-0097907 and CCR-0113181.

(JML). JML is a formal interface specification language for Java and has many fairly sophisticated features to facilitate writing abstract, precise, and complete behavioral descriptions of Java classes and interfaces [15, 16]. The runtime assertion checker generates Java bytecode from Java classes and interfaces with JML specifications. Runtime assertion checking is transparent in that, unless a specification assertion is violated, except for performance measures (time and space) the behavior of original program is unchanged.

Our main goal in developing a runtime assertion checker for JML is to leverage the power of formal interface specifications by providing an immediate and tangible payoff to programmers writing specifications in JML. This will contribute to making formal interface specifications practical and applicable in programming. Another goal is to provide the runtime assertion checker as a basis for other support tools such as specification-based automated testing and design by contract for Java [3].

JML is an extension to Java in the sense that it uses Java expressions for assertions. In this respect it follows the lead of Eiffel [19]. However, JML also incorporates many ideas and concepts from the model-oriented approach to specifications such as VDM [11] and Larch [9]. As thus, it offers many interesting challenges for implementing runtime assertion checker. These challenges include: notational complexity, several forms of quantifiers, specification inheritance [5] (through subclassing, interfaces, and refinement), specification-only (model) fields [17], and visibility control that is separate from Java's visibility control. In addition, the use of expressions in assertions raises the problem of undefinedness of assertions.

1.1. Related Work

Eiffel is a landmark programming language that integrates executable specifications into the language and its tools [19, 20]. The tools feature sophisticated ways to turn assertions on and off, and several levels of checking. Meyer's writings about Eiffel and its approach helped popularize the the concept of *design by contract* (*DBC*). DBC is rooted in formal methods, but because Eiffel uses expressions of the programming language in assertions, it is less intimidating (for many programmers) than languages that use lots of special-purpose mathematical notation, like Z [23].

However, Eiffel has several disadvantages for our purposes. First, it makes complete specifications more difficult because it does not allow specification-only (model) fields in objects, and because it does not support specification-only methods. It also does not feature a collection of built-in classes that represent immutable versions of collections that are useful in specifications, such as sets, sequences and tables. (Of course, Eiffel has such types, but they do not have immutable objects, and so are dangerous to use in specifications due to the possibility of side-effects.) Eiffel also does not permit the use of quantifiers in assertions, which are sometimes important for giving more complete specifications. Relatively complete specifications are needed if the specifications are to fulfill their role as test oracles [3].

Eiffel's popularization of DBC partly contributed to the availability of similar facilities in other programming languages, including Java. We know of several DBC tools for Java [1, 6, 7, 12, 13]. The approaches vary from a simple assertion mechanism similar to the C `assert` macro to full-fledged contract enforcement capabilities. Jass [1] and iContract [13] focus on the practical use of DBC in Java. Handshake and jContractor focus on implementation techniques such as library-based on-the-fly instrumentation of contracts [6, 12]. Contract Java focuses on properly blaming contract violations [7, 8]. These tools suffer from the same problems as Eiffel; that is, none of them has support for specification-only fields and methods, complete quantifiers, and they are not built around a set of immutable types designed for specification. They thus make it difficult to write complete specifications that are useful as test oracles.

A different kind of specification-based technique for detecting errors in programs is static analysis. An interesting example of such a tool is ESC/Java [4]. ESC/Java uses a theorem prover to find possible errors such as null pointer dereferences and array bounds violations. ESC/Java actually uses a subset of the JML specification notation but does not come with a runtime assertion checker, and so cannot be used in unit testing.

1.2. Outline

In the following sections, we describe our solutions to these challenges, and their rationale. We begin by describing how the runtime assertion checker deals with potential undefinedness and with quantifiers in JML's extension to Java's expression language. Section 3 describes how the runtime assertion checker deals with the sophisticated parts of JML's method specifications, and gives an overview of the compilation of specifications into runtime checks. Section 4 describes the strategy used to support specification inheritance in the runtime assertion checker. Section 5 describes how the checker supports specification-only model features of specifications. Finally, Section 6 offers a discussion and some conclusions.

2. Expressions and Assertions

2.1. Undefinedness

Assertions in JML are written in an extension to a side-effect free subset of Java expressions. The use of Java expressions in assertions leads to problems with potential undefinedness, for example, when an expression throws an exception. The JML's semantics for undefinedness is to substitute an arbitrary expressible value (of the correct type) for an undefined expression [16]. The challenge is to make the runtime assertion checker's handling of undefinedness faithful to the semantics of JML.

The runtime assertion checker has a unified framework for handling undefinedness in assertions, caused by various reasons such as exceptions, runtime errors, and non-executable specifications. We call our approach a *local, contextual interpretation*. It is a *local* interpretation in that an occurrence of an exception is interpreted locally — only by the smallest boolean subexpression that covers the exception-throwing expression. This supports abstract reasoning with standard logical laws; for example, the order of assertions in a conjunction or disjunction does not matter.

Our approach is also *contextual* in that, if a subexpression throws an exception, then the value used by the runtime assertion checker is determined by the subexpression's context. Since the runtime assertion checker wants to help detect errors, its goal is to falsify the overall assertion in which this subexpression occurs, whenever possible (while respecting the rules of logic). Falsifying the overall assertion allows the runtime assertion checker to signal an assertion violation to the user, and since the rules of logic are always respected, this assertion violation is a real violation that would otherwise go undetected.

To accomplish this we use both positive and negative contexts. To start with the overall assertion is considered to be in a positive context; in a positive context the checker tries to make an assertion false when it has the opportunity (because of exceptions from sub-expressions). In a negative context the checker tries to make assertions true. Negative contexts occur within expressions such as `!(`E`)`, where the polarity of the sub-expression E is reversed from that of the surrounding context. For example, assuming the surrounding context is positive, in the assertion `!(a[i] == null)`, the subexpression `(a[i] == null)` is in a negative context; thus if `a[i]` throws a null pointer exception, the checker uses `true` as the value of the subexpression `(a[i] == null)`, which makes the overall assertion false. Contexts for subexpressions are recursively defined according to the operators used.

Figure 1 shows the translation rule for universally quantified expressions that shows the contextual interpretation of undefinedness (see Section 2.2 for explanation). In our notation $[\![E, r, p]\!]$ denotes the translation — a piece of Java source code — that evaluates the expression E and stores the result into a variable named r. The translation is done in a context with polarity goal p, which is either `false` (for positive contexts) or `true` (otherwise). As shown, if an exception other than `JMLNonExecutableException` occurs while executing the translated code, the result is set to the current polarity goal.

```
[[(\forall $\vec{T}$ $\vec{v}$;$E_1$;$E_2$),$r$,$p$]] $\stackrel{\text{def}}{=}$
  try {
    $r$ = true;
    Iterator $i_1$ = $Q_1$.iterator();
    while ($r$ && $i_1$.hasNext()) {
      $T_1$ $v_1$ = ($T_1$)$i_1$.next();
      ...
      Iterator $i_n$ = $Q_n$.iterator();
      while($r$ && $i_n$.hasNext()) {
        $T_n$ $v_n$ = ($T_n$)$i_n$.next();
        [[$E_1$ ==> $E_2$, $r$, $p$]]
      }
      ...
    }
  }
  catch (JMLNonExecutableException e) {
    $r$ = !$p$;
  }
  catch (java.lang.Exception e) {
    $r$ = $p$;
  }
```

Figure 1. An example translation rule.

An interesting side-benefit of our contextual interpretation is that the checker is able to determine whether some assertions are executable at runtime, instead of requiring such determinations to be made statically. This flexibility is implemented by having the non executable subexpression throw a `JMLNonExecutableException`; when that happens, the result variable is set to the opposite of the current polarity goal, to make the overall assertion true if possible. The reason for this is that, since the actual value of a non-executable assertion is unknown, the checker must be conservative. Without knowing the polarity goal of the context, however, it would be impossible to allow this determination to be made at runtime.

2.2. Quantified Expressions

The runtime assertion checker supports all three forms of JML's quantified expressions: *universal* and *existential quantifiers* (`\forall` and `\exists`), *generalized quantifiers* (`\sum`, `\product`, `\min`, and `\max`), and a *numeric quantifier* (`\num_of`).

An extensible framework is provided to host different approaches to evaluating quantified expressions. In the current implementation, two evaluation strategies are supported: a *pattern-based static analysis approach* and *type-extension-based approach*. In the first approach, the runtime assertion checker statically analyzes a quantified expression and determines the set of objects or values that are necessary to decide the result of the quantification. (In the translation rule in Figure 1, this set is denoted as Q_i for quantified variable v_i.) The expression part is executed at runtime with the quantified variable bound to each element of the set. For a quantification over an integral type, patterns for intervals are identified. For example, the quantified expression `(\sum int x; x > 1 && x < 5; x)` defines the interval between 1 and 5, with both ends excluded, and thus can be evaluated by the runtime assertion checker. For a quantification over a reference type, patterns for collections are identified. For example, the quantified expression `(\forall Student p; ta.contains(p) || ra.contains(p); p.credits() <= 12))` defines a set consisting of elements of the collections `ta` and `ra`, and thus can be evaluated by the runtime assertion checker. If the static analysis fails to determine an interval or a collection, then the runtime assertion checker resorts to the extension-based approach.

In the extension-based approach, a quantified expression is evaluated for each "existing" object of the quantified type. The runtime assertion checker can generate code for automatically managing type extensions. One technique is to use *reference objects*, which hold on to referents (other objects)[1].

In addition to quantified expressions, the runtime assertion checker supports JML's set comprehension notation, which has a similar flavor to quantified expressions. This notation is implemented in a similar way to quantifiers.

3. Method Specifications

JML provides a rich set of syntactic sugars for specifying behaviors of methods — multiple clauses, nested specifications, case analysis, etc. [14, 16]. We desugar method specifications into a form that consists of only one occurrence of each kind of specification clause; for example the desugared specification only contains one `requires` clause (which is the method's precondition) [21]. A desugared method specification is used to generate several *assertion check methods*. The original method becomes a private method with a new name; it is replaced by a *wrapper method* that is automatically generated. The wrapper method delegates client method calls to the original method with appropriate assertion checks.

- *Precondition check methods*. When called, a precondition check method evaluates a method's precondition and throws a precondition violation exception if it does not hold.

- *Postcondition check methods*. When called, a postcondition check method evaluates the postcondition of the method and throws a postcondition violation exception if it does not hold. There are two kinds of postcondition check methods, *normal* and *exceptional postcondition check methods*. The first checks the normal postcondition specified by the `ensures` clause, and the second checks a method's exceptional postcondition specified by JML's `signals` clause.

Figure 2 shows a simplified skeleton of wrapper methods. The wrapper method first checks preconditions and class invariants, if any, in the pre-state by calling appropriate assertion check methods; as noted above, violations result in appropriate exceptions being thrown. If both assertions hold, the original method is invoked in a `try` statement and the return value, if exists, is saved into a local variable. The saved return value is used in checking the postcondition. The first `catch` block is for re-throwing an assertion violation detected during the execution of the original method[2]. If the control reaches the second `catch` block, it means that the execution of the original method has thrown an exception, thus the exceptional postcondition is checked by calling the exceptional postcondition check method. To make assertion checks transparent, the exceptional postcondition check method re-throws the original exception if the exceptional postcondition is satisfied. Finally, the `finally` block checks class-level post-state assertions such as invariants and history constraints[3].

In JML, an *old expression*, written as `\old(`e`)`, refers to the pre-state value of an expression, e. This is useful when specifying the behavior of a method with side effect [20]. The runtime assertion checker handles old expressions by evaluating them in the pre-state inside the precondition check method and caching the results in private fields. When the postcondition check method is called to evaluate expressions with old expressions, the corresponding private fields are used in place of the original old expressions.

An assertion can contain calls to methods with their own specifications. As in Eiffel, the runtime assertion checker recognizes this situation and does not check a method's assertions if the method is being called while checking assertions of another method (including the method itself).

4. Inheritance of Specifications

JML supports several ways of inheriting specifications: subclassing, interface extension and implementation, and

[1]A reference object is garbage collected at the discretion of the JVM, e.g., when the memory is low.

[2]In JML the assertion violation exceptions are organized into a hierarchy and the abstract class `JMLAssertionException` is the ultimate superclass of all assertion violation exceptions. Also, technically, precondition violation exceptions need to be translated into internal precondition violation exceptions as described in [3].

[3]A local flag is used to perform these checks only when no assertion violation such as precondition or postcondition violations has occurred.

```
T m(T1 x1, ..., Tn xn) {
  checkPre$m(x1, ..., xn);
  checkInv();
  T r = init_T; // initial value of T
  try {
    r = orig$m(x1, ..., xn);
    checkPost$m(x1, ..., xn, r);
  }
  catch (JMLAssertionException e) {
    throw e;
  }
  catch (Exception e) {
    checkXPost$m(x1, ..., xn, r, e);
  }
  finally {
    if (...) {
      checkInv();
      checkConstraint();
    }
  }
}
```

Figure 2. Simplified structure of wrapper methods.

refinement [5, 14, 16]. A subtype inherits from its supertypes such specifications as pre- and postconditions, type invariants, history constraints, etc.

The main difficulty implementing specification inheritance is that a subtype's assertion check method can't statically determine the existence of the corresponding assertion check method in its supertypes, due to separate compilation. That is, if the supertype was compiled without enabling runtime assertion checks, then its bytecode will not have the corresponding assertion check method.

4.1. Dynamic Delegation

To overcome this difficulty the runtime assertion checker implements specification inheritance using the Java's reflection facility. A call made using Java's reflection facility is termed a *dynamic call*. Thus a subtype's assertion check methods make dynamic calls to the corresponding assertion check methods of its supertype(s).[4] Dynamic calls are used even though the the names of supertypes and their assertion check methods are statically determined.

Thus, each assertion check method performs the following steps in order:

1. Check the assertions defined locally (i.e., in the given class), if they exist.

[4]Although Java has only single inheritance, specifications can also be inherited from interfaces.

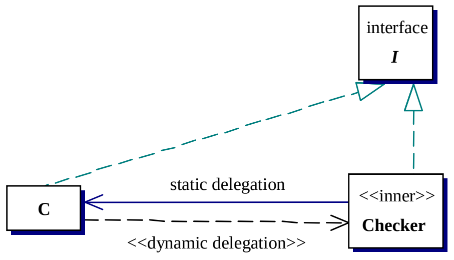

Figure 3. A class structure to support inheritance of interface specifications.

2. Check inherited assertions if any. For each statically determined pair of class and method names, perform the following steps by using the Java's reflection API.
   (a) Look up the assertion check method in the target class.
   (b) Invoke the target method, if it exists.
   (c) Combine the results appropriately, e.g., disjunction for preconditions and conjunction for class-level assertions such as invariants and constraints.
3. Report an assertion violation by throwing an assertion exception if the combined result becomes false.

To minimize runtime performance overhead due to dynamic delegation, the step 2 is performed only when necessary. For example, dynamic calls to pre- and postcondition checking methods are attempted only for a subtype's overriding methods; no dynamic delegation code is generated for the subtype's additional methods (those which do not override its supertype's methods).

4.2. Interface Specifications

In JML, specifications can also be written in interfaces. However, the checker cannot add an assertion check method directly to an interface because in Java an interface method must be abstract. For an interface, therefore, we generate a separate checker class, called a *surrogate class*, as a static inner class of the interface. The surrogate class is responsible for checking all the assertions specified in the interface.

A subtype of an interface inherits the interface's specifications by making dynamic calls to the assertion check methods defined in the interface's surrogate class. As shown in Figure 3, for specification inheritance, a class C implementing an interface I collaborates with I's surrogate class, I`.Checker`, in two ways:

- *Dynamic delegation.* To inherit assertions specified in the interface I, objects of the class C make dynamic calls to the corresponding assertion check methods defined in I's surrogate class, I`.Checker`, by creating surrogate objects as necessary. We often refer to these dynamic calls *up calls*.

- *Static delegation.* The surrogate class, I`.Checker`, implements the interface I by delegating to the corresponding methods of the class C. We refer to these static calls as *down calls*. The reason for implementing down calls is that assertions in the interface I may refer to methods specified in I. But the implementations of interface methods are found in the class C.

5. Model Specifications

JML provides several features that improve the level of abstraction in specifications. The ability to specify *model* elements such as model fields, model methods, and model types are the most distinguishing such feature of JML. Such model elements are only used in specifications; they cannot be used in normal Java programs. Model elements allow one to specify the abstract values of types [10]; these corresponds roughly to values at the conceptual (domain) level, and are abstract in the sense that one is not concerned with their (time or space) efficiency.

The runtime assertion checker can handle model fields provided that their correspondence to concrete fields (i.e., those in the actual Java program) is specified with an abstraction function. An abstraction function is specified with one form of JML's `represents` clause. For example, the following is a simple specification with a model field, `elems`, and an abstraction function.

```
import java.util.List;
public class Stack {
  private List contents;

  /*@ public model Object[] elems;
    @ public depends elems <- contents;
    @ private represents
    @  elems <- contents.toArray();
    @*/

  /*@ old int l = elems.length;
    @ requires e != null;
    @ modifies elems;
    @ ensures elems[0] == e &&
    @  elems.length == l + 1 &&
    @  (\forall int i; 0 <= i && i < i;
    @    elems[i+1] == \old(elems[i]));
    @*/
  public void push(Object e) { /* ... */ }
  /* ... */
}
```

A `represents` clause says how the value of a model variable is related to concrete program fields. (A `depends` clause allows such program fields to be modified when the model variable is modifiable [17].) This particular `represents` clauses says that the value of the model variable `elems` equals the expression `contents.toArray()`. Thus, given the concrete value of `contents`, the runtime assertion checker can calculate (retrieve) the abstract value of `elems`.

The runtime assertion checker generates a *model field access method* for each model field. A reference to a model field in assertions, e.g., `elems` in the specification of the method `push`, is replaced with a call to the corresponding model field access method. The default form of model field access methods throws a `JMLNonExecutableException`. However, an access method generated from a `represents` clause calculates an abstract value from concrete values and returns the result. For example, the above `represents` clause generates the following model field access method.

```
public Object[] model$elems() {
  return contents.toArray();
}
```

Model fields and `represents` clauses are also inherited by subtypes. The inheritance mechanism is similar to that of discussed in Section 4 and uses both dynamic and static forms of delegations. However, since `represents` clauses can be present in different types from where the corresponding model fields are declared, the implementation is more complex than we have space to explain here.

6. Discussion

It is important to give an informative message when an assertion is violated so that programmers can identify the source of the error. This is particularly true in JML because an assertion violation can be due to failures of assertions in many different places, sometimes in different files. For example, a postcondition violation may be caused by one of several `ensures` clauses or one of many inherited specifications (see Section 4). Similarly, a precondition violation occurs if all the `requires` clauses, perhaps inherited, fail to hold. The runtime assertion checker keeps track of the location information (file name, line number and column number) of assertions when desugaring and inheriting specifications. An assertion exception object stores a set of locations for constituents of the violated assertion in addition to the location of method call that causes the violation. It also stores the kind, the class name, and the method name of the violated assertion.

The use of dynamic calls to implement specification inheritance is modular and faithful to the JML's semantics.

The main drawback of this approach, however, is the runtime performance; it is slow due to the use of reflection facility. Tests reveal that the performance of dynamic method calls is marginally worse than that of static method calls, but introspection, i.e., dynamically looking up target classes and methods, is very costly. For example, our timing result shows that the dynamic inheritance approach is slower than the static approach (e.g., a textual copy approach) by a factor of 2.21 (or 221%). But, with a simple method lookup cache, the ratio drops to 1.25 (or 125%).

There are several areas for future work. One is to formalize the translation rules and to show the soundness of the translation with respect to the semantics of JML. Another is to improve the current implementation, for example, improving runtime efficiency and directly generating bytecode. We also plan to apply the runtime assertion checker in actual programming for empirical study of its effectiveness, and for developing other JML-based support tools to support unit testing [3] and design by contract for Java.

The JML runtime assertion checker described in this paper will be available from the JML web page at the URL: `http://www.jmlspecs.org`. An earlier implementation with many of these features is already available there.

References

[1] 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.

[2] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[3] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01-12a, Department of Computer Science, Iowa State University, Mar. 2002. To appear in ECOOP 2002.

[4] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.

[5] K. K. Dhara and G. 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, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[6] A. Duncan and U. Holzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA, Dec. 1998.

[7] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Lanugages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1–15, Oct. 2001.

[8] R. B. Findler, M. Latendresse, and M. 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), Spetember 10-14, 2001, Vienna, Austria*, Sept. 2001.

[9] J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

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

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

[12] M. Karaorman, U. Holzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. In P. 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.

[13] R. Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Kanguages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[14] G. T. Leavens and A. L. Baker. Enhancing the pre- and post-condition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. 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.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, Aug. 2001. See `www.jmlspecs.org`.

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

[18] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[19] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[20] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[21] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, Aug. 2001.

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

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