11-2001

# A Simple and Practical Approach to Unit Testing: The JML and JUnit Way

Yoonsik Cheon
*Iowa State University*

Gary T. Leavens
*Iowa State University*

# A Simple and Practical Approach to Unit Testing: The JML and JUnit Way

**Abstract**
Writing unit test code is labor-intensive, hence it is often not done as an integral part of programming. However, unit testing is a practical approach to increasing the correctness and quality of software; for example, the Extreme Programming approach relies on frequent unit testing. In this paper we present a new approach that makes writing unit tests easier. It uses a formal specification language's runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles. These oracles can be easily combined with hand-written test data. Instead of writing testing code, the programmer writes formal specifications (e.g., pre- and postconditions). This makes the programmer's task easier, because specifications are more concise and abstract than the equivalent test code, and hence more readable and maintainable. Furthermore, by using specifications in testing, specification errors are quickly discovered, so the specifications are more likely to provide useful documentation and inputs to other tools. We have implemented this idea using the Java Modeling Language (JML) and the JUnit testing framework, but the approach could be easily implemented with other combinations of formal specification languages and unit test tools.

**Keywords**
Unit testing, automatic test oracle generation, testing tools, runtime assertion checking, formal methods, programming by contract, JML language, JUnit testing framework, Java language

**Disciplines**
Software Engineering

# A Simple and Practical Approach to Unit Testing: The JML and JUnit Way

Yoonsik Cheon and Gary T. Leavens

TR #01-12
November 2001

**Keywords:** Unit testing, automatic test oracle generation, testing tools, runtime assertion checking, formal methods, programming by contract, JML language, JUnit testing framework, 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, JUnit; 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.

Submitted for publication

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

# A Simple and Practical Approach to Unit Testing: The JML and JUnit Way

Yoonsik Cheon and Gary T. Leavens

Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA 50011-1040, USA,
cheon@cs.iastate.edu, leavens@cs.iastate.edu,
+1-515-294-1580

**Abstract.** Writing unit test code is labor-intensive, hence it is often not done as an integral part of programming. However, unit testing is a practical approach to increasing the correctness and quality of software; for example, the Extreme Programming approach relies on frequent unit testing.

In this paper we present a new approach that makes writing unit tests easier. It uses a formal specification language's runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles. These oracles can be easily combined with hand-written test data. Instead of writing testing code, the programmer writes formal specifications (e.g., pre- and postconditions). This makes the programmer's task easier, because specifications are more concise and abstract than the equivalent test code, and hence more readable and maintainable. Furthermore, by using specifications in testing, specification errors are quickly discovered, so the specifications are more likely to provide useful documentation and inputs to other tools. We have implemented this idea using the Java Modeling Language (JML) and the JUnit testing framework, but the approach could be easily implemented with other combinations of formal specification languages and unit test tools.

## 1 Introduction

Program testing is an effective and practical way of improving correctness of software, and thereby improving software quality. It has many benefits when compared to more rigorous methods like formal reasoning and proof, such as simplicity, practicality, cost effectiveness, immediate feedback, understandability, and so on. There is a growing interest in applying program testing to the development process, as reflected by the Extreme Programming (XP) approach [3]. In XP, unit tests are viewed as an integral part of programming. Tests are created before, during, and after the code is written — often emphasized as "code a little, test a little, code a little, and test a little ..." [4]. The philosophy behind this is to use regression tests [25] as a practical means of supporting refactoring.

### 1.1 The Problem

However, writing unit tests is a laborious, tedious, cumbersome, and often difficult task. If the testing code is written at a low level of abstraction, it may be tedious and time-consuming to change it to match changes in the code. One problem is that there may simply be a lot of testing code that has to be examined and revised. Another problem occurs if the testing program refers to details of the representation of an abstract data type; in this case, changing the representation may require changing the testing program.

To avoid these problems, one should automate more of the writing of unit test code. The goal is to make writing testing code easier and more maintainable.

One way to do this is to use a framework that automates some of the details of running tests. An example of such a framework is JUnit [4]. It is a simple yet practical testing framework for Java classes; it encourages the close integration of testing with development by allowing a test suite be built incrementally.

However, even with tools like JUnit, writing unit tests often requires a great deal of effort. Separate testing code must be written and maintained in synchrony with the code under development, because the test class must inherit from the JUnit framework. This test class must be reviewed when the code under test changes, and, if necessary, also revised to reflect the changes. In addition, the test class suffers from the problems described above. The difficulty and expense of writing the test class are exacerbated during development, when the code being tested changes frequently. As a consequence, during development there is pressure to not write testing code and to not test as frequently as might be optimal.

We encountered these problems ourselves in writing Java code. The code we have been writing is part of a tool suite for the Java Modeling Language (JML). JML is a behavioral interface specification language for Java [29, 28]. In our implementation of these tools, we have been formally documenting the behavior of some of our implementation classes in JML. This enabled us to use JML's runtime assertion checker to help debug our code. In addition, we have been using JUnit as our testing framework. We soon realized that we spent a lot of time writing test classes and maintaining them. In particular we had to write many query methods to determine test success or failure. We often also had to write code to build expected results for test cases. We also found that refactoring made testing painful; we had to change the test classes to reflect changes in the refactored code. Changing the representation data structures for classes also required us to rewrite code that calculated expected results for test cases.

While writing unit test methods, we soon realized that most often we were translating method pre- and postconditions into the code in corresponding testing methods. The preconditions became the criteria for selecting test inputs, and the postconditions provided the properties to check for test results. That is, we turned the postconditions of methods into code for test oracles. A *test oracle* determines whether or not the results of a test execution are correct [37, 41, 45]. Developing test oracles from postconditions approach helped avoid dependence

of the testing code on the representation data structures, but still required us to write lots of query methods. In addition, there was no direct connection between the specifications and the test oracles, hence they could easily become inconsistent.

These problems led us to think about ways of testing code that would save us time and effort. We also wanted to have less duplication of effort between the specifications we were writing and the testing code. Finally, we wanted the process to help keep specifications, code, and tests consistent with each other.

### 1.2 Our Approach

In this paper, we propose a solution to these problems. We describe a simple and effective technique that automates the generation of oracles for unit testing classes. The conventional way of implementing a test oracle is to compare the test output to some pre-calculated, presumably correct, output [18, 35]. We take a different perspective. Instead of building expected outputs and comparing them to the test outputs, we monitor the specified behavior of the method being tested to decide whether the test passed or failed. This monitoring is done using the formal specification language's runtime assertion checker. We also show how the user can combine hand-written test inputs with these test oracles. Our approach thus combines formal specifications (such as JML) and a unit testing framework (such as JUnit).

Formal interface specifications include class invariants and pre- and postconditions. We assume that these specifications are fairly complete descriptions of the desired behavior. Although the testing process will encourage the user to write better preconditions, the quality of the generated test oracles will depend on the quality of the specification's postconditions.

We wrote a tool to automatically generate JUnit test classes from JML specifications. The generated test classes send messages to objects of the Java classes under test; they catch assertion violation exceptions from test cases that pass an initial precondition check. Such assertion violation exceptions are used to decide if the code failed to meet its specification, and hence that the test failed. If the class under test satisfies its interface specification for some particular input values, no such exceptions will be thrown, and that particular test execution succeeds. So the automatically generated test code serves as a test oracle whose behavior is derived from the specified behavior of the target class. (There is one complication which is explained in Section 4.) The user is still responsible for generating test data; however the generated test classes make it easy for the user to add test data.

### 1.3 Outline

The remainder of this paper is organized as follows. In Section 2 we describe the capabilities our approach assumes from a formal interface specification language and its runtime assertion checker, using JML as an example. In Section 3 we describe the capabilities our approach assumes from a testing framework, using

JUnit as an example. In Section 4 we explain our approach in detail; we discuss design issues such as how to decide whether tests fail or not, test fixture setup, and explain the automatic generation of test methods and test classes. In Section 5 we discuss how the user can add test data by hand to the automatically generated test classes. In Section 6 we describe our implementation of tools that support the design described in the previous sections. Finally, Section 7 describes related work and Section 8 concludes with a description of our experience, future plans, and the contributions of our work.

## 2 Assumptions About the Formal Specification Language

Our approach assumes that the formal specification language specifies the interface (i.e., names and types) and behavior (i.e., functionality) of classes and methods. We assume that the language has a way to express class invariants and method specifications consisting of pre- and postconditions.

Our approach can also handle specification of some more advanced features. One such feature is an *intra-condition*, usually written as an `assert` statement. Another is a distinction between normal and exceptional postconditions. A *normal postcondition* describes the behavior of a method when it returns without throwing an exception; an *exceptional postcondition* describes the behavior of a method when it throws an exception.

The Java Modeling Language (JML) [29, 28] is an example of such a formal specification language. JML specifications are tailored to Java, and its assertions are written in a superset of Java's expression language.

Fig. 1 shows an example JML specification. As shown, a JML specification is commonly written as annotation comments in a Java source file. Annotation comments start with `//@` or are enclosed in `/*@` and `@*/`. The `spec_public` annotation lets non-public declarations such as private fields `name` and `weight` be considered to be public for specification purposes[1]. The fourth line of the figure gives an example of an invariant, which should be true in each publicly-visible state.

In JML, preconditions start with the keyword `requires`, frame axioms start with the keyword `assignable`, normal postconditions start with the keyword `ensures`, and exceptional postconditions start with the keyword `signals` [20, 29, 28]. The semantics of such a JML specification states that a method's precondition must hold before the method is called. When the precondition holds, the method must terminate and when it does, the appropriate postconditions must hold. If it returns normally, then its normal postcondition must hold in the post-state (i.e., the state just after the body's execution), but if it throws an exception, then the appropriate exceptional postcondition must hold in the post-state. For example, the constructor must return normally when called with

[1] As in Java, a field specification can have an access modifier determining the visibility. If not specified, it defaults to that of the Java declaration; i.e., without the `spec_public` annotations, both `name` and `weight` could be used only in private specifications.

```
public class Person {
    private /*@ spec_public @*/ String name;
    private /*@ spec_public @*/ int weight;
    //@ public invariant name != null && !name.equals("") && weight >= 0;

    /*@ public behavior
      @   requires n != null && !n.equals("");
      @   assignable name, weight;
      @   ensures n.equals(name) && weight == 0;
      @   signals (RuntimeException e) false;
      @*/
    public Person(String n) { name = n; weight = 0; }

    /*@ public behavior
      @   assignable weight;
      @   ensures kgs >= 0 && weight == \old(weight + kgs);
      @   signals (IllegalArgumentException e) kgs < 0;
      @*/
    public void addKgs(int kgs) { weight += kgs; }

    /*@ public behavior
      @   ensures \result == weight;
      @   signals (RuntimeException e) false;
      @*/
    public /*@ pure @*/ int getWeight() { return weight; }

    /* ... */
}
```

**Fig. 1.** An example JML specification. The implementation of the method `addKgs` contains an error to be revealed in Section 5.2. This error was overlooked in our initial version of this paper, and so is an example of a "real" error.

a non-`null`, non-empty string `n`. It cannot throw a runtime exception because the corresponding exceptional postcondition is `false`.

JML has lots of syntactic sugar that can be used to highlight various properties for the reader and to make specifications more concise. For example, one can omit the `requires` clause if the precondition is `true`, as in the specification of `addKgs`. However, we will not discuss these sugars in detail here.

JML follows Eiffel [33, 34] in having special syntax, written `\old(`$e$`)` to refer to the pre-state value of $e$, i.e., the value of $e$ just before execution of the body of the method. This is often used in situations like that shown in the normal postcondition of `addKgs`.

For a non-`void` method, such as `getWeight`, `\result` can be used in the normal postcondition to refer to the return value. The method `getWeight` is specified to be *pure*, which means that its execution cannot have any side effects. In JML, only pure methods can be used in assertions.

In addition to pre- and postconditions, one can also specify intra-conditions with `assert` statements.

### 2.1 The Runtime Assertion Checker

The basic task of the runtime assertion checker is to execute code in a way that is transparent, unless an assertion violation is detected. That is, if a method is called and no assertion violations occur, then, except for performance measures (time and space) the behavior of the method is unchanged. In particular, this implies that, as in JML, assertions can be executed without side effects.

We do not assume that the runtime assertion checker can execute all assertions in the specification language. However, only the assertions it can execute are of interest in this paper.

We assume that the runtime assertion checker has a way of signaling assertion violations to a method's callers. In practice this is most conveniently done using exceptions. While any systematic mechanism for indicating assertion violations would do, to avoid circumlocutions, we will assume that exceptions are used in the remainder of this paper.

The runtime assertion checker must have some exceptions that it can use without interference from user programs. These exceptions are thus reserved for use by the runtime assertion checker. We call such exceptions assertion violation exceptions. It is convenient to assume that all such assertion violation exceptions are subtypes of a single assertion violation exception type.

JML's runtime assertion checker [6] can execute a constructive subset of JML assertions, including some forms of quantifiers. In functionality, it is similar to other design by contract tools [26, 33, 34, 42]; such tools could also be used with our approach.

To explain how JML's runtime checker monitors Java code for assertion violations, it is necessary to explain the structure of the instrumented code compiled by the checker. Each Java class and method with associated JML specifications is instrumented as shown in Fig. 2. The original method body is enclosed inside a `try` statement. To handle old expressions (as used in the postcondition of `addKgs`), the instrumented code evaluates each old expression occurring in the postconditions in the pre-state, and binds the resulting value to a locally-defined name. When the code to evaluate such a postcondition is executed, the corresponding locally-defined name is used in place of the old expression.

Next, the class invariant and precondition, if any, are evaluated and checked in the pre-state, i.e., before the execution of the original method body. The checking code throws either `JMLInvariantException` or `JMLPreconditionException` if these assertions are not satisfied in the pre-state. After the original method is executed in the `try` block, the exceptional postconditions are checked in the second `catch` block and the normal postcondition is checked in the `finally` block.[2] To make assertion checking transparent, the code that checks the exceptional

[2] Local variables are used to distinguish the normal return case from the case where an exception is being thrown.

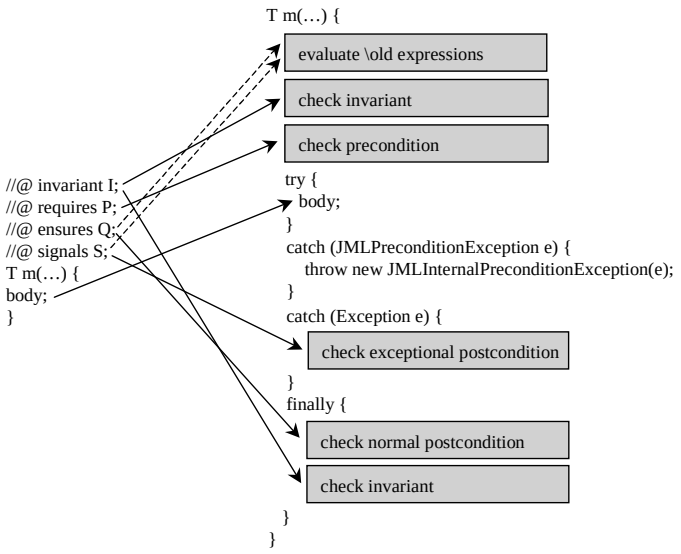


**Fig. 2.** A translation scheme for runtime assertion checking of JML specifications

postcondition re-throws the original exception if the exceptional postcondition is satisfied; otherwise, it throws a `JMLPostconditionException`. In the `finally` block, the class invariants are checked again, but this time in the post-state. The purpose of the first `catch` block is explained below (see Section 4.1).

Our approach assumes that the runtime assertion checker can distinguish two special kinds of assertion violation exceptions: precondition exceptions and internal precondition exceptions. Other distinctions among exception types are useful in reporting errors to the user, but are not important for our approach.

In JML the assertion violation exceptions are organized as shown in Fig. 3. The ultimate superclass of all assertion violation exceptions is the abstract class `JMLAssertionException`. This class has several subclasses that correspond to different kinds of assertion violations, such as precondition violations, postcondition violations, invariant violations, and so on. The precondition and internal precondition exceptions of our assumptions correspond to the types `JMLPreconditionException` and `JMLInternalPreconditionException`.

## 3 Assumptions About the Testing Framework

Our approach assumes that the unit testing framework has a way to run test methods. For convenience, we will assume that test methods can be grouped into test classes. A test method can indicate to the framework whether the test fails or succeeds.

We also assume that there is a way to provide test data to test methods. Following JUnit's terminology, we call this the test fixture. A *test fixture* is a

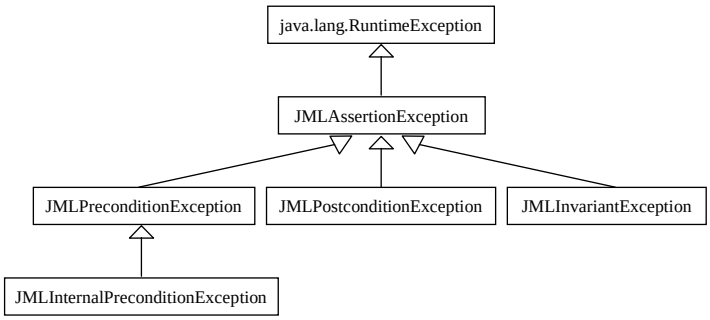


**Fig. 3.** A part of the exception hierarchy for JML runtime assertion violations

context for performing test execution; it typically contains several declarations for test inputs and expected outputs.

For the convenience of the users of our approach, we assume that it is possible to define a global test fixture, i.e., one that is shared by all test methods in a test class. With a global test fixture, one needs ways to initialize the test inputs, and to undo any side effects of a test after running the test.

JUnit is a simple, useful testing framework for Java [4, 23]. In JUnit, a test class consists of a set of test methods. The simplest way to tell the framework about the test methods is to name them all with names beginning with "`test`". The framework uses introspection to find all these methods, and can run them when requested.

Fig. 4 is a sample JUnit test class, which is designed to test the class `Person`. Every JUnit test class must be a subclass, directly or indirectly, of the framework class `TestCase`. The class `TestCase` provides a basic facility to write test classes, e.g., defining test data, asserting test success or failure, and composing test cases into a test suite.

One uses methods like `assertEquals`, defined in the framework, to write test methods, as in the test method `testAddKgs`. Such methods indicate test success or failure to the framework. For example, when the arguments to `assertEquals` are equal, the test succeeds; otherwise it fails. Another such framework method is `fail`, which signals test failure. JUnit assumes that a test succeeds unless the test method signals test failure, for example, by calling `fail`.

The framework provides two methods to manipulate the test fixture: `setUp` creates objects and does any other tasks needed to run a test, and `tearDown` undoes otherwise permanent side-effects of tests. For example, the `setUp` method in Fig. 4 creates a new `Person` object, and assigns it to the test fixture variable `p`. The `tearDown` method can be omitted if it does nothing. JUnit automatically invokes the `setUp` and `tearDown` methods before and after each test method is executed (respectively).

The static method `suite` creates a *test suite*, i.e., a collection of test methods. To run tests, JUnit first obtains a test suite by invoking the method `suite`, and then runs each test method in the suite. A test suite can contain several test

```
import junit.framework.*;
public class PersonTest extends TestCase {
  public PersonTest(String name) {
    super(name);
  }
  public void testAddKgs() {
    p.addKgs(10);
    assertEquals(10, p.getWeight());
  }
  private Person p;
  protected void setUp() {
    p = new Person("Baby");
  }
  protected void tearDown() {
  }
  public static Test suite() {
    return new TestSuite(PersonTest.class);
  }
  public static void main(String args[]) {
    String[] testCaseName = {PersonTest.class.getName()};
    junit.textui.TestRunner.main(testCaseName);
  }
}
```

**Fig. 4.** A sample JUnit test class

methods, and it can contain other test suites, recursively. Fig. 4 uses Java's reflection facility to create a test suite consisting of all the test methods of class `PersonTest`.

## 4 Test Oracle Generation

This section presents the details of our approach to automatically generating a JUnit test class from a JML-annotated Java class. We first describe how test success or failure is determined. Then we describe the convention and protocol for the user to supply test data to the automatically generated test oracles in the test classes. After that we discuss in detail the automatic generation of test methods and test classes.

### 4.1 Deciding Test Success or Failure

A test class has one test method, `test`$M$, for each method, $M$, to be tested in the original class. The method `test`$M$ runs $M$ on several test cases. Conceptually, a *test case* consists of a pair of a receiver object (an instance of the class being tested) and a sequence of argument values; for testing static methods, a test case does not include the receiver object.

Success or failure of a call to $M$ for a given test case is determined by whether the runtime assertion checker throws an exception during the $M$'s execution, and what kind of exception is thrown. If no exception is thrown, then the test case succeeds (assuming the call returns), because there was no assertion violation, and hence the call must have satisfied its specification.

Similarly, if the call to $M$ for a given test case throws an exception that is not an assertion violation exception, then this also indicates that the call to $M$ succeeded for this test case. Such exceptions are passed along by the runtime assertion checker because it is assumed to be transparent. Hence if the call to $M$ throws such an exception instead of an assertion violation exception, then the call must have satisfied $M$'s specification. With JUnit, such exceptions must, however, be caught by the test method `test`$M$, since any such exceptions are interpreted by the framework as signaling test failure. Hence, the `test`$M$ method must catch and ignore all exceptions that are not assertion violation exceptions.

If the call to $M$ for a test case throws an assertion violation exception, however, things become interesting. If the assertion violation exception is not a precondition exception, then the method $M$ is considered to fail that test case. However, we have to be careful with the treatment of precondition violations. A precondition is an obligation that the client must satisfy; nothing else in the specification is guaranteed if the precondition is violated. Therefore, when a test method `test`$M$ calls method $M$ and $M$'s precondition does not hold, we do not consider that to be a test failure; rather, when $M$ signals a precondition exception, it indicates that the given test input is outside $M$'s domain, and thus is inappropriate for test execution.

On the other hand, precondition violations that arise inside the execution of $M$ should still be considered to be test failures. To do this, we distinguish two kinds of precondition violations that may occur when `test`$M$ runs $M$ on a test case, $(o, \vec{x})$:

- The precondition of $M$ fails for $(o, \vec{x})$, which indicates, as above, that the test case $(o, \vec{x})$ is outside $M$'s domain.
- A method $f$ called from within $M$'s body signals a precondition violation, which indicates that $M$'s body did not meet $f$'s precondition, and thus that $M$ failed to correctly implement its specification on the test case $(o, \vec{x})$. (Note that if $M$ calls itself recursively, then $f$ may be the same as $M$.)

The JML runtime assertion checker converts the second kind of precondition violation into an internal precondition violation exception. Thus, `test`$M$ decides that $M$ fails on a test case $(o, \vec{x})$ if $M$ throws an internal precondition violation exception, but rejects the test case $(o, \vec{x})$ if it throws a (non-internal) precondition violation exception. This treatment of precondition exceptions was the main change that we had to make to JML's existing runtime assertion checker to implement our approach.

To summarize, the result of a test execution is a test failure if and only if an assertion violation exception other than a non-internal precondition violation is thrown on the invocation of the method being tested.

### 4.2 Setting Up Test Cases

In our approach, a test fixture is responsible for constructing test data, i.e., constructing receiver objects and argument objects. For example, testing the method `addKgs` of class `Person` (see Fig. 1) requires one object of class `Person` in the test fixture, and one value of type `int`. The first object will be the receiver of the message `addKgs`, and the second will be the argument. In our approach, a test fixture does not need to construct expected outputs, because success or failure is determined by observing the runtime assertion checker, not by comparing results to expected outputs.

How does the user define these objects and values in the test fixture, for use by the automatically generated test methods? There are three general approaches to answering this question: (i) one can define a separate fixture for each test method, (ii) a global test fixture shared by all methods, or (iii) a combination of both. In the first approach, each test method defines a separate set of test fixture variables, resulting in more flexible and customizable configuration. However this makes more work for the user. Thus, we take the second approach. Other than execution time, there is no harm to run a test execution with a test fixture of another method if both are type-compatible; it may turn out to be more effective. The more test cases, the better! One might worry that some of these inputs may violate preconditions for some methods; recall, however, that (non-internal) precondition violations are not treated as test failures, and so such inputs do not cause any problems.

Our implementation defines test fixture variables as one-dimensional arrays. The test fixture variables are defined to be `protected` fields of the test class so that users can initialize them in subclasses of the automatically generated test classes. To let test fixtures be shared by all test methods, we adopt a simple naming convention. Let $C$ be a class to be tested and $T_1, T_2, \ldots, T_n$ be the formal parameter types of all the methods to be tested in the class $C$. Then, the test fixture for the class $C$ is defined as[3]:

$C$`[] testee;` $T_1$`[] v`$T_1$`; ... ;` $T_n$`[] v`$T_n$`;`

The first array named `testee` refers to the set of receiver objects and the rest are for argument objects.

If a particular method has formal parameter types $A_1, A_2, \ldots, A_m$, where each $A_i$ is drawn from the set $\{T_1, \ldots, T_n\}$, then its test cases are:

$$\{\langle \texttt{testee}[i], \texttt{v}A_1\texttt{[}j_1\texttt{]}, \ldots, \texttt{v}A_m\texttt{[}j_m\texttt{]}\rangle \mid 0 \le i < \texttt{testee.length}, \\ 0 \le j_1 < \texttt{v}A_1\texttt{.length}, \ldots, 0 \le j_m < \texttt{v}A_m\texttt{.length}\}$$

[3] For array types we use the character `$` to denote their dimensions, e.g., `vint_$_$` for `int[][]`.

For example, the methods to be tested from the class `Person` (that are shown in Fig. 1) have only one type of formal parameter, which is the type `int` (in the method `addKgs`). Therefore, the class `Person`'s fixture is defined as follows:

```
protected Person[] testee; // receiver objects
protected int[] vint;      // argument objects
```

So the test cases for the method `addKgs` are

$$\{\langle \texttt{testee[}i\texttt{]}, \texttt{vint[}j\texttt{]} \rangle \mid 0 \leq i < \texttt{testee.length}, 0 \leq j < \texttt{vint.length}\}$$

whereas those of the method `getWeight`, are

$$\{\langle \texttt{testee[}i\texttt{]} \rangle \mid 0 \leq i < \texttt{testee.length}\}.$$

The test fixture variables such as `testee` and `vint` will be initialized and reset by the methods `setUp` and `tearDown` respectively (see Section 4.4 for more details).

### 4.3 Test Methods

Recall that there will be a separate test method, `test`$M$ for each target method, $M$, to be tested. The purpose of `test`$M$ is to determine the result of calling $M$ with each test case and to give an informative message if the test execution fails for that test case. The method `test`$M$ accomplishes this by invoking $M$ with each test case and checking whether the runtime assertion checker throws an assertion violation exception or not.

To describe our implementation, let $C$ be a Java class annotated with a JML specification and $C$`_JML_Test` the JUnit test class generated from the class $C$. For each instance (i.e., non-`static`) method of the form:

$T$ $M$`(`$A_1$ $a_1$`,...,` $A_n$ $a_n$`) throws` $E_1$`,...,` $E_m$ `{ /* ... */ }`

of the class $C$, a corresponding test method `test`$M$ is generated in the test class $C$`_JML_Test`.

The generated test method `test`$M$ has the code skeleton shown in the Fig. 5. The test method first initializes a sequence of local variables to the test fixture variables corresponding to the formal parameters of the method under test. The local variables are named the same as the formal parameters of the method under test. The local variables are not necessary, but they make the resulting code more comprehensible if one should ever try to read the generated test class. For each test case, given by the test fixture, the test method then invokes the method under test in a `try` statement and sees if the JML runtime assertion checker throws an exception. As described above, an internal precondition exception or a `JMLAssertionException` that is not a precondition exception means a failure of the test execution; thus an appropriate error message is composed and printed. This message will contain the failed method name, the failed test case (i.e., the values of receiver and argument objects), and the exception thrown by the JML runtime assertion checker.

```
public void testM() {
  final A1[] a1 = vA1;
  ...
  final An[] an = vAn;
  for (int i0 = 0; i0 < testee.length; i0++)
    for (int i1 = 0; i1 < a1.length; i1++)
      ...
      for (int in = 0; in < an.length; in++) {
        try {
          if (testee[i0] != null) {
            testee[i0].M(a1[i1], ..., an[in]);
          }
        }
        catch (JMLInternalPreconditionException e) {
          String msg = /* a String showing the test case */;
          fail(msg + NEW_LINE + e.getMessage());
        }
        catch (JMLPreconditionException e) {
          continue;   // success for this test case
        }
        catch (JMLAssertionException e) {
          String msg = /* a String showing the test case */;
          fail(msg + NEW_LINE + e.getMessage());
        }
        catch (java.lang.Exception e) {
          continue;  // success for this test case
        }
    }
}
```

**Fig. 5.** A skeleton of generated test methods

A similar form of test method is generated for testing the static methods of a class. For static methods, however, test messages are sent to the class itself, therefore, the outer-most `for` loop is omitted and the body of the `try` block is replaced with $C.M(a_1[i_1], \ldots, a_n[i_n])$.

A test method is generated only for the public, protected, and package visible methods of the class to be tested. That is, no test methods are generated for private methods, although these may be indirectly tested through testing of the other methods. Also, test methods are not generated for a `static public void` method named `main`; testing the main method seems inappropriate for unit testing. Furthermore, no test methods are generated for a class's constructor methods. However, we should note that constructor methods are implicitly tested when the test fixture variable `testee` is initialized; if the pre- or postconditions of a constructor are violated during initialization, the test setup will fail, and the user will be led to the error by the message in the assertion violation exception.

Fig. 6 is an example test method generated for the method `addKgs` of the class `Person`. We use a very simple convention to name the generated test methods. We prefix the original method name with the string "`test`" and capitalize the initial letter of the method name.[4]

```
public void testAddKgs() {
  final int[] kgs = vint;
  for (int i = 0; i < self.length; i++)
    for (int j = 0; j < kgs.length; j++) {
      try {
        if (testee[i] != null) {
          testee[i].addKgs(kgs[j]);
        }
      }
      catch (JMLInternalPreconditionException e) {
        String msg = /* a String showing the test case */;
        fail(msg + NEW_LINE + e.getMessage());
      }
      catch (JMLPreconditionException e) {
        continue;
      }
      catch (JMLAssertionException e) {
        String msg = /* a String showing the test case */;
        fail(msg + NEW_LINE + e.getMessage());
      }
      catch (java.lang.Exception e) {
        continue;
      }
   }
}
```

**Fig. 6.** Code generated for testing the method `addKgs` of the class `Person`. The actual code for generating the error messages `msg` is lengthy, and so is suppressed here.

### 4.4 Test Classes

In addition to test fixture definition and test methods described in the previous sections, a JUnit test class must have several other methods. Let $C$ be a Java class annotated with a JML specification and $C$`_JML_Test` the JUnit test class generated from the class $C$.

[4] To prevent clashes among the test method names for overloaded methods, the tool actually appends a suffix representing the formal parameter types to each test method name. The suffix consists of each parameter types separated by `$_`, e.g., `testAddKgs$_int` for `addKgs(int)`). We ignore these details in the rest of the paper.

A JUnit test class should be a subclass of the framework class `TestCase`, directly or indirectly. The `package` and `import` definitions for $C$`_JML_Test` are copied verbatim from the class $C$. As a result, the generated test class will reside in the same package as in the original testee class. This allows the test class to have access to package-private members of the testee class. In addition to the copied import definitions, several `import` statements are generated that import JUnit-specific packages.

The test class includes several boilerplate methods that are the same in all the generated test classes. A constructor, a main method, and a method for test suites are automatically generated, as shown in Fig. 7.

```
public C_JML_Test(String name) {
  super(name);
}
public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
}
public static Test suite() {
    return new TestSuite(C_JML_Test.class);
}
```

**Fig. 7.** Boilerplate methods for JUnit test class for testing class $C$

As explained in the previous section, test fixture variables are defined as protected member fields. A test fixture definition is accompanied by default `setUp` and `tearDown` methods. The `setUp` method is supposed to be redefined by a subclass to populate the test fixture with actual test data. Let $T_1, T_2, \ldots, T_n$ be the formal parameter types of all the methods to be tested in the class $C$. Then, the test fixture for the class $C$ and the `setUp` and `tearDown` methods are defined as in Fig. 8 (see also Section 4.2).

For example, the generated code for the test fixture definition and test fixture setup methods of the test class, `Person_JML_Test`, generated for our example class `Person`, will be as shown in Fig. 9.

Test fixture variables are initialized to zero-element arrays so that the test class can be run "out of the box," although it is not very useful to do so. At least the `setUp` method, and sometimes the `tearDown` method, will need to be to be overridden in the subclasses to populate the fixture with actual test data (see Section 5).

## 5 Supplying and Running Test Cases

To perform actual test executions, the user must supply reasonable test inputs by initializing the test fixture's variables. This can be done either by directly editing the `setUp` method of the generated test class or by subclassing and redefining

```
protected C[] testee;
protected T1[] vT1;
...
protected Tn[] vTn;

protected void setUp() {
  testee = new C[0];
  vT1 = new T1[0];
  ...
  vTn = new Tn[0];
}
protected void tearDown() {
}
```

**Fig. 8.** Skeleton of the automatically generated test fixture for a test class

```
protected Person[] testee;
protected int[] vint;

protected void setUp() {
  testee = new Person[0];
  vint = new int[0];
}
protected void tearDown() {
}
```

**Fig. 9.** Example of an automatically generated test fixture, for `Person`

the `setUp` method. We recommend the subclassing approach. The subclassing approach prevents the user from losing test cases if the test class is regenerated. In addition, in the subclass the user can also tune the testing by adding hand-written test methods. The JUnit framework collects and exercises the added test methods together with the automatically generated methods.

### 5.1 Populating the Test Fixture with Test Data

A test input can be any type-correct value. For example, we can set the test fixture variables for the class `Person` as written in Fig. 10. Remember that the test class for the class `Person` has two test fixture variables `testee` and `vint`, of types `Person[]` and `int[]` respectively.

As shown, test inputs can even be `null`. Also there can be aliasing among the test fixture variables, although this is not shown in our example. With the above test fixture, the `addKgs` method is tested 24 times, one for each pair of `testee[`$i$`]` and `vint[`$j$`]`, where $0 \leq i < 4$ and $0 \leq j < 6$.

```
import junit.framework.*;
import junit.extensions.*;

public class PersonTestCase extends Person_JML_Test
{
  public PersonTestCase(String name) {
    super(name);
  }

  protected void setUp() {
    testee = new Person[4];
    testee[0] = new Person("Baby");
    testee[1] = new Person("Cortez");
    testee[2] = new Person("Isabella");
    testee[3] = null;
    vint = new int[] { 10, -22, 0, 1, 55, 3000 };
  }

  public static Test suite() {
    return new TestSuite(PersonTestCase.class);
  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

**Fig. 10.** The user-defined class that defines the test fixture for the class `Person`

### 5.2 Running Test Cases

It is very simple to perform test execution with user-defined test cases such as the class `PersonTestCase` shown in Fig. 10. It is done in three steps as follows.

1. Generate and compile an instrumented Java class for the class to be tested (e.g., class `Person`).
2. Generate and compile a JUnit test class for the target class (e.g., class `Person_JML_Test`).
3. Compile and run the user-defined test case class (e.g., class `PersonTestCase`).

The first two steps are done using the JML support tools (see Section 6 for details); for example, the command "`jtest --all Person.java`" does the first two steps, assuming that the class `Person` is stored in the file `Person.java`. The last step can be done with any Java compiler and interpreter.

Fig. 11 shows the result of running the test cases of Fig. 10, i.e., the class `PersonTestCase`. It reveals the error that we mentioned in the caption of Fig. 1.

As the above output shows, one test failure occurred for the method `addKgs`. The test data that caused the failure is also printed, i.e., the receiver, an object of class `Person` with name `Baby`, and the argument of value `-24`.

```
Time: 0.02
There was 1 failure:
1) testAddKgs$_int(PersonTestCase)junit.framework.AssertionFailedError:
        Message 'addKgs' to object testee[0] of value: Person("Baby",-12)
        Argument kgs (vint[1]) of value: -22
        Failed POSTCONDITION of method Person.addKgs at Person.java:19
        at Person_JML_Test.testAddKgs$_int(Person_JML_Test.java:81)
        t PersonTestCase.main(PersonTestCase.java:20)
FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

**Fig. 11.** Output from running the tests in `PersonTestClass`

A corrected implementation of the method `addKgs` is shown in Fig. 12. (Compare this with the specification and the faulty implementation shown in Fig. 1.)

```
public void addKgs(int kgs) {
    if (kgs >= 0)
       weight += kgs;
    else
       throw new IllegalArgumentException("Negative Kgs");
}
```

**Fig. 12.** Corrected implementation of method `addKgs` in class `Person`

.

## 6 Implementation

We have implemented our approach as part of the JML support tool-set. The implementation of the test oracle generator has the structure shown in Fig. 13. It follows the familiar *pipe-and-filter* architectural style [44]. The implementation is based on the existing input and output facilities of JML for the internal program representation. The test oracle generator module receives a type-annotated abstract syntax tree (TAST) from the JML type checker, transforms it into a JUnit test class by directly mutating the internal representation (AST), and finally outputs an external textual representation (i.e., Java source code) by using the unparser.

Adding the test oracle generator into the JML toolset was rather straightforward. Each module is written as a tree-walk using ANTLR [36]. Hence we only needed to write one ANTLR grammar file that transforms the input TAST into the output AST of a JUnit class. We were able to do this without much difficulty by reusing existing utility and helper classes. We also had to slightly

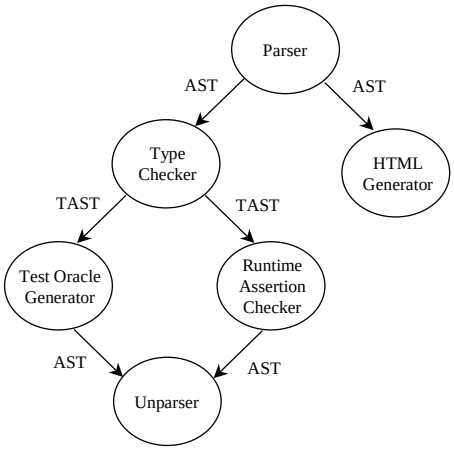


**Fig. 13.** The architecture of JML toolset with the test oracle generator

modify and adjust the main program to reflect new command options and flags and properly hook up the oracle generator module.

We also provide two shell scripts, `jmlc` and `jtest`, to further facilitate the use of JML and JUnit in combination. The first compiles JML annotated Java source code into bytecode with runtime assertion checking enabled. The second compiles JML annotated Java source classes into both assertion check-enabled Java classes and JUnit test classes. Both shell scripts make the code generation process transparent to the user by directly writing Java bytecode from source files.

## 7 Related Work

There are now quite a few runtime assertion checking facilities developed and advocated by many different groups of researchers. One of the earliest and most popular approaches is Meyer's view of Design By Contract (DBC) implemented in the programming language Eiffel [32–34]. Eiffel's success in checking pre- and postconditions and encouraging the DBC discipline in programming partly contributed to the availability of similar facilities in other programming languages, including C [42], C++ [12, 16, 39, 46], Java [2, 13, 14, 24, 26], .NET [1], Python [38], and Smalltalk [7]. These approaches vary widely from a simple assertion mechanism similar to the C `assert` macros, to full-fledged contract enforcement capabilities. Among all that we are aware of, however, none uses its assertion checking capability as a basis for automated program testing. Thus, our work is unique in the DBC community in using a runtime assertion checking to automate program testing.

Another difference between our work and that of other DBC work is that we use a formal specification language, JML, whose runtime assertion checker

supports manipulation of abstract values. As far as we know, all other DBC tools work only with concrete program values. However, in JML, one can specify behavior in terms of abstract (specification) values, rather than concrete program values [6, 28, 29]. So-called *model variables* — specification variables for holding not concrete program data but their abstractions — can be accompanied by `represents` clauses [28]. A `represents` clause specifies an abstraction function (or relation) that maps concrete values into abstract values. This abstraction function is used by the runtime assertion checker in JML to manipulate assertions written in terms of abstract values.

The traditional way to implement test oracles is to compare the result of a test execution with a user supplied, expected result [18, 35]. A test case, therefore, consists of a pairs of input and output values. In our approach, however, a test case consists of only input values. And instead of directly comparing the actual and expected results, we observe if, for the given input values, the program under test satisfies the specified behavior. As a consequence, programmers are freed from not only the burden of writing test programs, often called *test drivers*, but also from the burden of pre-calculating presumably correct outputs and comparing them. The traditional schemes are constructive and direct whereas ours is behavior observing and indirect.

Several researchers have already noticed that if a program is formally specified, it should be possible to use the specification as an oracle [37, 41, 45]. Thus, the idea of automatically generating test oracles from formal specifications is not new, but the novelty lies in employing a runtime assertion checker as the test oracle engine. This aspect seems to be original and first explored in our approach. Peters and Parnas discussed their work on a tool that generates a test oracle from formal program documentation [37]. The behavior of program is specified in a relational program specification using tabular expressions, and the test oracle procedure, generated in C++, checks if an input and output pair satisfies the relation described by the specification. Their approach is limited to checking only pre and postconditions, thus allowing only a form of black-box tests. In our approach, however we also support *intra-conditions*, assertions that can be specified and checked within a method, i.e., on internal states [28]; thus our approach supports a form of white-box tests. As mentioned above, our approach also support abstract value manipulation. In contrast to other work on test oracles, our approach also support object-oriented concepts such as specification inheritance.

There is a large volume of research papers published on the subject of formal specification-based software testing [5, 8, 9, 21, 25, 41, 43]. Most of these papers are concerned with methods and techniques for automatically generating test cases from formal specifications, though there are some addressing the problem of automatic generation of test oracles as noted before [37, 41, 45]. A general approach is to derive the so-called *test conditions*, a descriptions of test cases, from the formal specification of each program module [8]. The derived test conditions can be used to guide test selection and to measure comprehensiveness of an existing test suite, and sometimes they even can be turned into executable forms

[8, 9]. The degree of support for automation varies widely from the derivation of test cases, to the actual test execution and even to the analysis of test results [9, 41]. Some approaches use existing specification languages [19, 21], and others have their own (specialized) languages for the description of test cases and test execution [8, 9, 41, 43]. All of these works are complimentary to the approach described in this paper, since, except as noted above, they solve the problem of defining test cases which we do not attempt to solve, and they do not solve the problem of easing the task of writing test oracles, which we partially solve.

## 8 Conclusion and Future Work

We presented a simple but effective approach to implementing test oracles from formal behavioral interface specifications. The idea is to use the runtime assertion checker as the decision procedure for test oracles. We have implemented this approach using JML, but other runtime assertion checkers can easily be adapted to work with our approach. The only complication is that the runtime assertion checker has to distinguish two kinds of precondition violations: those that arise from the call to a method and those that arise within the implementation of the method; the first kind of precondition violations is used to reject test cases as not being applicable to the call, while the second indicates a test failure.

Our approach trades the effort one might spend in writing code to construct expected test outputs for effort spent in writing formal specifications. Formal specifications are more concise and abstract than code, and hence we expect them to be more readable and maintainable. Formal specifications also serve as more readable documentation than testing code, and can be used as input to other tools such as extended static checkers [10].

Most testing methods do not check behavioral results, but focus only on defining what to test. Because most testing requires a large number of test cases, manually checking test results severely hampers its effectiveness, and makes repeated and frequent testing impractical. To remedy this, our approach automatically generates test oracles from formal specifications, and integrates these test oracles with a testing framework to automate test executions. This helps make our implementation practical. It also makes our approach a blend of formal verification and testing.

In sum, the main goal of our work —to ease the writing of testing code— has been achieved.

A main advantage of our approach is the improved automation of testing process, i.e., generation of test oracles from formal behavioral interface specifications and test executions. We expect that, due to the automation, writing test code will be easier. Indeed, this has been our experience. However, measuring this effect is future work.

Another advantage of our approach is that it helps make formal methods more practical and concretely usable in programming. One aspect of this is that test specifications and target programs can reside in the same file. We expect that this will have a positive effect in maintaining consistency between test

specifications and the programs to be tested, although again this remains to be empirically verified.

A third advantage is that our approach can achieve the effect of both black-box testing and white-box testing. White-box testing can be achieved by specifying intra-conditions, predicates on internal states in addition to pre- and post-conditions. Assertion facilities such as the `assert` statement are an example of intra conditions; they are widely used in programming and debugging. JML has several specification constructs for specifying intra-conditions which support white-box testing.

Finally, in our approach a programmer may extend and add his own testing methods to the automatically generated test oracles. This can be done easily by adding hand-written test methods to a subclass of the automatically generated test class.

Our approach frees the programmer from writing unit test code, but the programmer still has to supply actual test data by hand. In the future, we hope to partially alleviate this problem by automatically generating some of test inputs from the specifications. There are several approaches proposed by researchers to automatically deriving test cases from formal specifications. It would be very exciting to apply some of the published techniques to JML. JML has some features that may make this future work easier, in particular various forms of specification redundancy. In JML, a *redundant* part of a specification does not itself form part of the specification's contract, but instead is a formalized commentary on it [27]. One such feature are formalized examples, which can be thought of as specifying both test inputs and a description of the resulting post-state. However, for such formalized examples to be useful in generating test data, they would: (a) have to be specified constructively, and (b) it would have to be possible to invert the abstraction function, so as to build concrete representation values from them.

Another area of future work is to gain more experience with our approach. The application of our approach so far has been limited to the development of the JML support tools themselves, but our initial experience seems very promising. We were able to perform testing as an integral part of programming with minimal effort and to detect many kinds of errors. Almost half of the test failures that we encountered were caused by specification errors; this shows that our approach is useful for debugging specifications as well as code. However, we have yet to perform significant, empirical evaluation of the effectiveness of our approach.

JML and a version of the tool that implements our approach can be obtained through the JML web page at the following URL:

`http://www.cs.iastate.edu/~leavens/JML.html`

## Acknowledgments

The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea, and by

grants CCR-0097907 and CCR-0113181 from the US National Science Foundation. Thanks to Curtis Clifton and Markus Lumpe for comments on an earlier draft of this paper.

## References

1. Karine Arnout and Raphael Simon. The .NET contract wizard: Adding design by contract to languages other than Eiffel. In *Proceedings of TOOLS 39, 29 July -3 August 2001, Santa Barbara, California*, pages 14–23. IEEE Computer Society, 2001.
2. 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.
3. Kent Beck. *Extreme Programming Explained.* Addison-Wesley, 2000.
4. Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
5. Gilles Bernot, Marie Claude Claudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.
6. Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report TR #00-08, Department of Computer Science; Iowa State University, Ames, IA, May 2000.
7. Manuela Carrillo-Castellon, Jesus Garcia-Molina, Ernesto Pimentel, and Israel Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 9(7):23–28, November/December 1996.
8. Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *Proceedings of ISSTA 96, San Diego, CA*, pages 62–70. IEEE Computer Society, 1996.
9. J. L. Crowley, J. F. Leathrum, and K. A. Liburdy. Isues in the full scale use of formal methods for automated testing. *ACM SIGSOFT Software Engineering Notes*, 21(3):71–78, May 1996.
10. 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.
11. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.
12. Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115, Portland, OR, August 1992. USENIX Assoc. Berkeley, CA, USA.
13. 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.
14. Robert Bruce Findler and Matthias Felleisen. Behavioral interface contracts for Java. Technical Report CS TR00-366, Department of Computer Science, Rice University, Houston, TX, August 2000.
15. David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1994.
16. Pedro Guerreiro. Simple support for design by contract in C++. In *Proceedings of TOOLS 39, 29 July -3 August 2001, Santa Barbara, California*, pages 24–34. IEEE Computer Society, 2001.

17. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.
18. R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
19. Teruo Higashino and Gregor v. Bochmann. Automatic analysis and test case derivation for a restricted class of LOTOS expressions with data parameters. *IEEE Transactions on Software Engineering*, 20(1):29–42, January 1994.
20. 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.
21. Pankaj Jalote. Specification and testing of abstract data types. *Computing Languages*, 17(1):75–82, 1992.
22. Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J., second edition, 1990.
23. JUnit. Http://www.junit.org.
24. 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.
25. Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of ISSTA 98, Clearwater Beach, FL*, pages 143–152. IEEE Computer Society, 1998.
26. Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Kanguages and Systems, Los Alamitos, California*, pages 295–307, 1998.
27. Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. Davies J.M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume II*, volume 1708 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, September 1999.
28. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See `www.cs.iastate.edu/~leavens/JML.html`.
29. Gary T. Leavens, Albert L. Baker, and Clye Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.
30. Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
31. Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).
32. B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.
33. Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

34. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.
35. D.J. Panzl. Automatic software test driver. *IEEE Computer*, pages 44–50, April 1978.
36. T.J. Parr and R. W. Quong. ANTLR: A predicate-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, July 1995.
37. Dennis Peters and David L. Parnas. Generating a test oracle from program documentation. In *Proceedings of ISSTA 94, Seattle, Washington, August, 1994*, pages 58–65. IEEE Computer Society, August 1994.
38. Reinhold Plosch and Josef Pichler. Contracts: From analysis to C++ implementation. In *Proceedings of TOOLS 30*, pages 248–257. IEEE Computer Society, 1999.
39. Sara Porat and Paul Fertig. Class assertions in C++. *Journal of Object-Oriented Programming*, 8(2):30–37, May 1995.
40. Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, August 2001.
41. Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of ISSTA 94, Seattle, Washington, August, 1994*, pages 138–152. IEEE Computer Society, August 1994.
42. David R. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
43. Sriram Sankar and Roger Hayes. ADL: An interface definition language for specifying and testing software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994. Proceedings of the Workshop on Interface Definition Language, Jeannette M. Wing (editor), Portland, Oregon.
44. Mary Shaw and David Garlan. *Software Architecture*. Prentice Hall, Inc., 1996.
45. P. Stocks and D. Carrington. Test template framework: A specification-based test case study. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 11–18. IEEE Computer Society, June 1993.
46. David Welch and Scott Strong. An exception-based assertion mechanism for C++. *Journal of Object-Oriented Programming*, 11(4):50–60, July/August 1998.