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

Yoonsik Cheon and Gary T. Leavens

TR #01-12c
November 2001; revised March 2002, May 2003, and September 2003.

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.

This version is modified from that appearing in the European Conference on Object-Oriented Programming (ECOOP), Málaga, Spain, June 10-14, 2002. *Lecture Notes in Computer Science*, Copyright © Springer-Verlag, 2002.

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,leavens}@cs.iastate.edu

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, Extreme Programming 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 the 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 [26] 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) [7]. JML is a behavioral interface specification language for Java [30, 29]. 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 [6, 10, 11]. 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 [35, 38, 41]. 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 [20, 34]. 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. The quality of these postconditions is the user's responsibility, just as the quality of hand-written test oracles would be.

We wrote a tool, `jmlunit`, 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 discuss other issues. In Section 7 we describe related work and we conclude, in Section 8, 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, 30] 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 `@*/`. Within the latter kind of comment, at-signs (`@`) on the beginning of lines are ignored. 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, method specifications precede the corresponding method declarations. Method 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` [22, 29, 30]. 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

[1] As in Java, a field specification can have an access modifier determining its visibility. If not specified, the visibility defaults to the visibility 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.length() > 0 && weight >= 0;

    /*@ public behavior
      @   requires n != null &&  name.length() > 0;
      @   assignable name, weight;
      @   ensures n.equals(name) && weight == 0;
      @   signals (Exception 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 (Exception 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 5.1. This error was overlooked in our initial version of this paper, and so is an example of a "real" error.

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 a non-`null`, non-empty string `n`. It cannot throw an 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 [32, 33] 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 can execute a constructive subset of JML assertions, including some forms of quantifiers [6, 10, 11]. In functionality, it is similar to other design by contract tools [27, 32, 33, 39]; 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 by example in Fig. 2. The original method becomes a private method, e.g., `addKgs` becomes `internal$addKgs`. The checker generates a new method, e.g., `addKgs`, to replace it, which calls the original method, `internal$addKgs`, inside a `try` statement.

The generated method first checks the method's precondition and class invariant, if any.[2] If these assertions are not satisfied, this check throws either `JMLEntryPreconditionError` or `JMLInvariantError`. After the original method

[2] To handle old expressions (as used in the postcondition of `addKgs`), the instrumented code evaluates each old expression occurring in the postconditions from within the `checkPre$addKgs` method, and binds the resulting value to a private field of the class. The corresponding private field is used when checking postconditions.

```
public void addKgs(int kgs) {
  checkPre$addKgs(kgs); // check precondition
  checkInv(); // check invariant
  boolean rac$ok = true;
  try {
    internal$addKgs(kgs);
    checkPost$addKgs(kgs); // check normal postcondition
  } catch (JMLEntryPreconditionError e) {
    rac$ok = false;
    throw new JMLInternalPreconditionError(e);
  } catch (JMLAssertionError e) {
    rac$ok = false;
    throw e;
  } catch (Throwable e) {
    try { // check exceptional postcondition
      checkExceptionalPost$addKgs(kgs, e);
    } catch (JMLAssertionError e1) {
      rac$ok = false; // an exceptional postcondition violation
      throw e1;
    }
  } finally {
    if (rac$ok) {
      checkInv(); // check invariant
    }
  }
}
```

Fig. 2. The top-level of the run-time assertion checker's translation of the `addKgs` method in class `Person`. (Some details have been suppressed.)

is executed in the `try` block, the normal postcondition is checked, or, if exceptions were thrown, the exceptional postconditions are checked in the third `catch` block. To make assertion checking transparent, the code that checks the exceptional postcondition re-throws the original exception if the exceptional postcondition is satisfied; otherwise, it throws a `JMLNormalPostconditionError` or `JMLExceptionalPostconditionError`. In the `finally` block, the class invariant is checked again. 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 kinds of precondition assertion violations: *entry precondition violations* and *internal precondition violations*. The former refers to violations of preconditions of the method being tested. The latter refers to precondition violations that arise during the execution of the tested method's body. Other distinctions among assertion violations are useful in reporting errors to the user, but are not important for our approach.

In JML the assertion violation exceptions are organized into an exception hierarchy as shown in Fig. 3. The ultimate superclass of all assertion violation exceptions is the abstract class `JMLAssertionError`. This class has several subclasses that correspond to different kinds of assertion violations, such as precondition violations, postcondition violations, invariant violations, and so on. Our assumed entry precondition and internal precondition violations are realized by the types `JMLEntryPreconditionError` and `JMLInternalPreconditionError`. Both are concrete subclasses of the abstract class `JMLPreconditionError`.

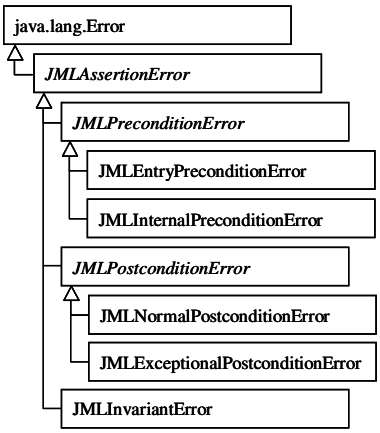

Fig. 3. A part of the exception hierarchy for JML runtime assertion violations.

3 Assumptions About the Testing Framework

Our approach assumes that unit tests are to be run for each method of each class being tested. We assume that the framework provides *test methods*, which are methods used to test the methods of the class under test. For convenience, we will assume that test methods can be grouped into test classes.

In our approach, each test method executes several test cases for the method it is testing. Thus we assume that a test method can indicate to the framework whether each test case fails, succeeds, or was meaningless. The outcome will be *meaningless* if an entry precondition violation exception occurs for the test case; details are given in Section 5.1.

We also assume that there is a way to provide test data to test methods. Following JUnit's terminology, we call this a test fixture. A *test fixture* is a context for executing a test; it typically contains several declarations for variables that hold test inputs. The fixture may also contain declarations for variables holding 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, 24]. 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 methods into a test suite.

```
import junit.framework.*;
public class PersonTest extends TestCase {
  private Person p;
  public PersonTest(String name) {
    super(name);
  }
  public void testAddKgs() {
    p.addKgs(10);
    assertEquals(10, p.getWeight());
  }
  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.

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 not equal, the test fails. Another such framework method is `fail`, which directly indicates test failure. JUnit assumes that a test succeeds unless the test method throws an exception or indicates test failure. Thus the only way a test method can indicate success is to return normally.

JUnit thus does not provide a way to indicate that a test execution was meaningless. This is because it is geared toward counting executions of test methods instead of test cases, and because hand-written tests are assumed to be meaningful. However, in our approach we need to extend JUnit to allow the counting of test case executions and to track which test cases were meaningful. We extended the JUnit framework to do this by writing a class `JMLTestRunner`, which tracks the meaningful test cases executed.

JUnit provides two methods to manipulate the test fixture: the `setUp` method creates objects and does any other tasks needed to run a test, and the `tearDown` method 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`. Both methods can be omitted if they do 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 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 outcomes are 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 Outcomes

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* for an instance method 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 and constructors, a test case does not include the receiver object.

The outcome of a call to M for a given test case is determined by whether the runtime assertion checker throws an exception during 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, specifically, its exceptional postcondition. 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. We call the outcome of such a test execution "meaningless" instead of calling it either a success or failure. 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, tc:

- The precondition of M fails for tc, which indicates, as above, that the test case tc is outside M's domain. As noted earlier, this is called an *entry* precondition violation.
- 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 tc. (Note that if M calls itself recursively, then f may be the same as M.) Such an assertion violation is an *internal* precondition violation.

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 tc if M throws an internal precondition violation exception, but rejects the test case tc as meaningless if it throws an entry 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. The treatment of meaningless test case executions is also the only place where we had to extend the JUnit testing framework.

To summarize, the outcome of a test execution is "failure" if an assertion violation exception other than an entry precondition violation is thrown, is "meaningless" if an entry precondition violation is thrown, and "success" otherwise.

4.2 Setting Up Test Cases

In our approach, various initialization methods are responsible for constructing test data, i.e., constructing receiver objects and argument objects. For example,

a test case for the method `addKgs` of class `Person` (see Fig. 1) requires one object of type `Person` 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, there is no need to construct expected outputs, because success or failure is determined by observing the runtime assertion checker, not by comparing results to expected outputs.

Where does the user define the test data that are used in the generated test methods? There are several possibilities:

- *Separate test fixture.* Each test method has a separate set of test fixture variables, resulting in a very flexible and customizable configuration. However, defining such fixture variables becomes complicated and requires more work from the user.
- *Global test fixture.* All test methods share the same set of test fixture variables. The approach is simple and intuitive, and thus defining fixture variables requires less work from the user. However, the user has less control in that, because of shared fixture variables, it becomes hard to supply specific test cases to specific test methods.
- *Combination.* Some combination of the above two approaches, which has a simple and intuitive test fixture configuration, and yet to gives the user more control.

Our earlier work [12] adopted the global test fixture approach. The rationale was that the more test cases would be the better and the simplicity of use would outweigh the benefit of more control. There would be no harm to run test methods with test cases of other methods (if test cases are type-compatible). Some of test cases might violate the precondition; however, entry precondition violations are not treated as test failures, and so such test cases cause no problems. However, our experience showed that more control was sometimes necessary.[3] Thus a combination approach is used in the current implementation.

In the combination approach we adopted, a global test fixture array variable is declared for each formal parameter type. In addition, a global fixture array variable is declared for the receivers. Thus, as in the global fixture approach, the fixture variables are shared by all test methods, and all test methods share the receiver array. Methods with the same parameter type share the same fixture array for arguments of that type. However, if a method has two arguments of the same type, our test drivers maintain (potentially) separate arrays for each argument, which can avoid interference between test data for different arguments.

The fixture variables for a particular test method are those that correspond to the method's parameter types plus the receiver, and the test cases for the method are all possible combinations of the array elements found in these fixture variables. The test fixture variables are declared as `protected` fields of the test class so that users can initialize them in subclasses of the automatically

[3] Also, in our earlier implementation, all of the test fixture variables were reinitialized for each call to the method under test, which resulted in very slow testing. Most of this reinitialization was unnecessary.

generated test classes (see Section 5 for details). To let test fixtures be shared by all test methods, we adopt a simple naming convention. Aside from the name "`receivers`," which holds the receiver objects for instance methods, a fixture variable's name is the name of its type prefixed by the character `v`[4], e.g., `vint` for type `int`. Thus, if C is a class to be tested and $T_1, T_2, \ldots, T_n$ are the formal parameter types of all the methods to be tested in the class C, then, the test fixture for the class C is:

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

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

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

For example, the class `Person` will have the following test fixture variables; its methods have only two formal parameter types, `String` in the constructor and `int` in the method `addKgs`.

```
protected Person[] receivers;
protected String[] vString;
protected int[] vint;
```

The set of test cases for the method `addKgs` is thus:

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

whereas the set of test cases for the method `getWeight` is:

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

In addition to the test fixture variables, the combination approach generates a pair of initialization and uninitialziation methods for each test fixture variable[5] (see Figure 5). The user must override the `init_receivers` and `init_v`T methods to initialize the corresponding test fixture array variables, thus supplying test cases to test methods (see Section 5 for details). The user can also override the corresponding `uninit_` methods to undo any permanent side-effects from the use of such variables, but usually this is not necessary. The generated test

[4] For an array type, the character `$` is used to denote its dimension, e.g., `vint_$_$` for `int[][]`. Also to avoid name clashes, the `jmlunit` tool uses fully qualified names for reference types; for example, instead of `vString`, the actual declaration would be for a variable `vjava_lang_String`. To save space, we do not show the fully qualified names.

[5] We thank David Cok for pointing out problems with the earlier approach and discussing such an extension.

methods call the initialization and uninitialziation methods of its test fixture variables before and after each test execution (respectively, see Section 4.3 for more details).

The name of method under test is passed as an argument to these initialization and uninitialization methods. This gives the user some primitive control over what test data is used for each testing method. For example, the user can avoid test data that would cause a time-consuming method to run for a long time by using different test data for just that method.

```
protected abstract void init_receivers(String forMethodName);
protected void uninit_receivers(String forMethodName) {}
protected abstract void init_vString(String forMethodName);
protected void uninit_vString(String forMethodName) {}
protected abstract void init_vint(String forMethodName);
protected void uninit_vint(String forMethodName) {}
```

Fig. 5. Test fixture methods for the class `Person`.

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 outcome 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 indicating test failure when the runtime assertion checker throws an assertion violation exception that is not an entry precondition violation. Test methods also note when test cases were rejected as meaningless.

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 Fig. 6.

The code for this method uses local variables with the same names as M's formal parameters. These local variables are used to avoid the sharing of mutable objects between distinct formal parameters with the same type; in such a case the initialization method should reinitialize the array on each call, and the assignment of the array to the second variable of the same type will thus receive a fresh copy of the array. These local variables may also make the resulting code more comprehensible than using the test fixture names, which are based solely on the types of the formal parameters.

```
public void testM() {
  this.init_receivers("M");
  for (int i0 = 0; i0 < receivers.length; i0++) {
    this.init_vA1("M");
    final A1[] a1 = vA1;
    for (int i1 = 0; i1 < a1.length; i1++) {
      ...
      this.init_vAn("M");
      final An[] an = vAn;
      for (int in = 0; in < an.length; in++) {
        if (receivers[i0] == null) {
          /* ... tell framework test case was meaningless ... */
        } else {
          try {
            receivers[i0].M(a1[i1], ..., an[in]);
          }
          catch (JMLEntryPreconditionError e) {
            /* ... tell framework test case was meaningless ... */
            continue;
          }
          catch (JMLAssertionError e) {
            String msg = /* a String showing the test case */;
            fail(msg + NEW_LINE + e.getMessage());
          }
          catch (java.lang.Throwable e) {
            continue; // success for this test case
          }
        }
      }
      this.uninit_vAn("M");
      ...
    }
    this.uninit_vA1("M");
  }
  this.uninit_receivers("M");
}
```

Fig. 6. A skeleton of generated test methods.

The nested `for` loops in Fig. 6 loop over all test cases. Each loop iterates over a single array corresponding to the receiver or to a formal parameter. Before each such loop the corresponding `init_` method is invoked with the name of the method being tested; following each such loop the corresponding `uninit_` method is invoked, also with the name of the method being tested. Inside the innermost loop, the method being tested itself is called, using the test case given by using the surrounding loop variables to index into the arrays corresponding to the formal parameters. For each such test case, 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 assertion violation exception (`JMLAssertionError`) other than an entry precondition violation exception means a failure of the test execution; thus an appropriate error message is composed and printed. The message contains 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.

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

By default, the `jmlunit` tool only generates test methods for public methods in the class or interface being tested. It is impossible to test private methods from outside the class, but users can tell the tool whether they would like to also test protected and package visible methods. Also, test methods are not generated for a `static public void` method named `main`; testing the main method seems inappropriate for unit testing.

Fig. 7 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.[6]

4.4 Test Classes

In addition to test fixture definition, the text fixture initialization and uninitialization methods, and the test methods described in the previous sections, the generated JUnit test class has several other methods. These are described in this section.

Let C be a Java class or interface annotated with a JML specification and C`_JML_Test` the JUnit test class generated from the class C.

As a JUnit test class, C`_JML_Test` inherits from the JUnit framework's class `TestCase`. The `package` and `import` definitions for C`_JML_Test` are copied verbatim from the type C. As a result, the generated test class will reside in the same package. This allows the test class to access package-visibility members of the class under test.

[6] If necessary, the tool appends a unique suffix to prevent a name clash due to method overloading.

```
public void testAddKgs() {
  this.init_receivers("addKgs");
  for (int i = 0; i < receivers.length; i++) {
    this.init_vint("addKgs");
    final int[] kgs = vint;
    for (int j = 0; j < kgs.length; j++) {
      if (receivers[i] == null) {
         /* ... tell framework test case was meaningless ... */
      } else {
        try {
          receivers[i].addKgs(kgs[j]);
        }
        catch (JMLEntryPreconditionError e) {
          /* ... tell framework test case was meaningless ... */
          continue;
        }
        catch (JMLAssertionError e) {
          String msg = /* a String showing the test case */;
          fail(msg + NEW_LINE + e.getMessage());
        }
        catch (java.lang.Throwable e) {
         continue;
        }
      }
   }
}
```

Fig. 7. Code generated for testing the method `addKgs` of the class `Person`. Details of generating the error messages and telling the framework about meaningless test cases are suppressed.

The test class includes several boilerplate methods that are the same in all the generated test classes. These consist of a method used to check on the initialization of test fixture variables, a method used to check that the code under test was compiled by JML's runtime assertion checker, and a constructor, as shown in Fig. 8.

The first boilerplate method, `test$FixtureInitialization`, is intended to catch errors in which the user-supplied `init_` methods do not properly initialize variables in the test fixture. This method calls each of the `init_` methods and tests that they each initialize the corresponding fixture variable to a non-empty array. This is something we added to the `jmlunit` tool as we gained experience with it, because we found that when we added new methods to the type being tested, the tool would sometimes add new fixture variables and the we would

```
public void test$FixtureInitialization() {
  this.init_receivers("init_receivers");
    if (receivers == null) {
      junit.framework.Assert.fail("Fixture variable "
        + "‘receivers’" + " was not initialized by init_receivers.");
    }
    if (receivers.length == 0) {
      junit.framework.Assert.fail("Fixture variable "
        + "‘receivers’" + " has no data elements.");
    }
    ...
}
public void test$IsRACCompiled() {
  junit.framework.Assert.assertTrue("code for class C"
      + " was not compiled with jmlc"
      + " so no assertions will be checked!",
    org.jmlspecs.jmlrac.runtime.JMLChecker.isRACCompiled(C.class) );
}
public C_JML_Test(String name) {
  super(name);
}
protected void setUp() { }
protected void tearDown() { }
```

Fig. 8. Boilerplate methods for JUnit test class for testing class C.

forget to initialize them properly.[7] If a fixture variable has zero length, the corresponding for loop that uses it does not execute its body, and thus no tests are done for methods using its type; hence this check is important in preventing a user from thinking that testing has been adequate when in fact it has not. With the check, any such errors are caught and the tests cannot pass unless all fixture variables are initialized.

The second boilerplate method, `test$IsRACCompiled` checks to make sure that the type being tested was compiled with JML's assertion checking compiler. This prevents the user from thinking that all tests have passed when in fact no testing was done because no assertions were checked.

5 Supplying and Running Test Cases

To perform actual test executions, the user must provide a subclass that overrides the `init_` methods, thus providing the test data that initializes the test fixture variables. A subclass of the test class is used so that the user does not lose the test

[7] The compiler catches errors that occur when there are more initializations than fixture variables generated by the `jmlunit` tool; this can happen when the user deletes a method from a class.

data when the test class is regenerated. In addition, the user can tune the testing by adding hand-written test methods to this subclass. The JUnit framework collects and exercises the added test methods together with the automatically generated methods when the user runs the subclass.

The `jmlunit` tool produces a skeleton subclass of the test case class automatically, if one does not exist. For testing a type C, the subclass is called C`_JML_TestCase`. E.g., the subclass for `Person` is called `Person_JML_TestCase`, and is shown in Fig. 9.

Test inputs are supplied by overriding the `init_` methods for each test fixture variable. The responsibility of each method, `init_v`T is to initialize the protected array variable, `v`T, with elements of type T for use as test inputs to the method whose name is passed to it in the formal parameter `methodName`.

As can be seen in Fig. 9, a test input can be any type-correct value. For example, we can initialize the test fixture variables for the class `Person` by overriding the corresponding `init_` methods as shown in the figure. Recall that the test class for the class `Person` has three test fixture variables: `receivers`, `vString`, and `vint`, of types `Person[]`, `String[]`, and `int[]`, respectively. With the initialization methods shown in the figure, the `addKgs` method is tested 18 times, one for each pair of `receivers[`i`]` and `vint[`j`]`, where $0 \leq i < 3$ and $0 \leq j < 6$.

As shown by the initialization of `vString`, test inputs can even be `null`. As might be expected, we have found that the use of `null` as a test input is very helpful in making sure the preconditions of methods that take reference types as parameters protect the method against null pointer exceptions.

It is possible, and sometimes desirable, that there is aliasing among the test fixture variables, although this is not shown in our example. This can be arranged by using private variables and by having some initialization methods invoke others.

5.1 Running Test Cases

It is very simple to perform test execution with user-defined test cases such as the class `Person_JML_TestCase` shown in Fig. 9. This is done in the following steps.

1. Generate instrumented Java byte code for the type, C to be tested using the `jmlc` script; e.g., `jmlc Person.java`.
2. Generate a JUnit test class, C`_JML_Test`, and, if necessary, a skeleton of the JUnit test case class, C`_JML_TestCase`, using the `jmlunit` script; e.g., `jmlunit Person.java`.
3. Write initialization code in the JUnit test case class, C`_JML_TestCase`; e.g., write code for `init_receivers`, `init_vString`, and `init_vint` in the test case class `Person_JML_TestCase.java`.
4. Compile the user-defined test and test case classes using a Java compiler (other than `jmlc`); e.g., `javac Person_JML_Test*.java`.

```
public class Person_JML_TestCase extends Person_JML_Test
{
  public Person_JML_TestCase(java.lang.String name) {
    super(name);
  }

  public static void main(java.lang.String[] args) {
    org.jmlspecs.jmlunit.JMLTestRunner.run(suite());
  }

  public static junit.framework.Test suite() {
    return new junit.framework.TestSuite(Person_JML_TestCase.class);
  }

  protected void init_receivers(java.lang.String methodName) {
    receivers
      = new Person[] {
        new Person("Baby"),
        new Person("Cortez"),
        new Person("Isabella"),
      };
  }

  protected void init_vjava_lang_String(java.lang.String methodName) {
    // elements of an immutable type only need to be initialized once
    if (vjava_lang_String == null) {
      vjava_lang_String
         = new java.lang.String[] {
           null,
           "Baby",
           "Martin",
           "Martina",
         };
    }
  }

  protected void init_vint(java.lang.String methodName) {
    // elements of an immutable type only need to be initialized once
    if (vint == null) {
      vint
         = new int[] {
           0, 10, -22, 1, 55, 3000,
         };
    }
  }
}
```

Fig. 9. The user-defined class that defines the test fixture for the class `Person`, `Person_JML_TestCase`.

5. Run the test case class's tests, using the JML runtime assertion checking libraries, as provided by the `jml-junit` script or the `jmlrac` script; e.g., `jml-junit Person_JML_TestCase` or `jmlrac Person_JML_TestCase`.

Fig. 10 shows the result of running the test cases of Fig. 9, i.e., the class `Person_JML_TestCase`. It reveals the error that we mentioned in the caption of Fig. 1.

```
% jmlrac Person_JML_TestCase
....F..
Time: 0.141
There was 1 failure:
1) testAddKgs(Person_JML_TestCase)junit.framework.AssertionFailedError:
        Method 'addKgs' applied to
        Receiver receivers[0]: Person("Baby",-12)
        Argument 'kgs' (vint[2]): -22
        Caused org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError
        by method Person.addKgs regarding specifications at
Person.java:16:25 when
        '\old(weight + kgs)' is -12
        'kgs' is -22
        'this' is Person("Baby",-12)
        at Person_JML_Test.testAddKgs(Person_JML_Test.java:199)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(...)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(...)
        at Person_JML_Test.run(Person_JML_Test.java:22)
        at org.jmlspecs.jmlunit.JMLTestRunner.doRun(JMLTestRunner.java)
        at org.jmlspecs.jmlunit.JMLTestRunner.run(JMLTestRunner.java)
        at Person_JML_TestCase.main(Person_JML_TestCase.java:17)

FAILURES!!!
Tests run: 6,  Failures: 1,  Errors: 0

JML test runs: 12/13 (meaningful/total)
```

Fig. 10. Output from running the tests in `Person_JML_TestCase`.

As the output in the figure 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 `-22`. More information is printed in the assertion violation message by JML's runtime assertion checker, including the file, line number, and column number where the violated assertion is located.

A corrected implementation of the method `addKgs` is shown in Fig. 11. (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. 11. Corrected implementation of method `addKgs` in class `Person`.

To report the numbers of test successes or failures, the framework counts the number of test methods. This is not the right measure in our approach, because each test method runs the method under test with all possible combinations of test data. To get more accurate report, one can use our specialized test runner class, `JMLTestRunner`, instead of a JUnit's test runner class such as `junit.framework.textui.TestRunner`. (The `JMLTestRunner` is invoked automatically from the main method of the generated test case class, see Fig. 9). The class `JMLTestRunner` reports the number of meaningful test runs and the total number of test runs in terms of test data, as shown in the last line of Fig. 10. Such a report prevents the user from having a wrong impression that the class under test satisfied all tests when in fact no test has actually be executed due to all test cases being inapplicable.[8]

6 Discussion

What should the outcome of a test case be if a method detects an invariant is violated in at the beginning of a method's execution? Such a situation can arise if clients can directly write an object's fields, or if aliasing allows clients to manipulate the object's representation without calling its methods. The question is whether such invariant violations should be treated as a test failure or as a rejection of the test data (i.e., as a "meaningless" test). One reason for rejecting the test data is that one can consider the invariant to be part of the precondition. One may also consider an object malformed if it does not satisfy the invariant. However, treating such violations as if the test case were meaningless seems to mask the underlying violation of information hiding, and so our current implementation treats these as test failures.

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 [31–33]. Eiffel's success in checking pre- and

[8] We thank an anonymous referee for pointing out this problem.

postconditions and encouraging the DBC discipline in programming partly contributed to the availability of similar facilities in other programming languages, including C [39], C++ [16, 19, 37, 42], Java [2, 17, 18, 25, 27], .NET [1], Python [36], and Smalltalk [8]. 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, 29, 30]. So-called *model variables* — specification variables for holding not concrete program data but their abstractions — can be accompanied by `represents` clauses [13, 29]. 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 [10].

The traditional way to implement test oracles is to compare the result of a test execution with a user supplied, expected result [20, 34]. 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 [35, 38, 41]. 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 [35]. 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 [29]; 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 or-

acles, our approach also supports object-oriented concepts such as specification inheritance.

There are many research papers published on the subject of testing using formal specifications [5, 9, 14, 23, 26, 38, 40]. 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 [35, 38, 41]. A general approach is to derive so-called *test conditions*, a description of test cases, from the formal specification of each program module [9]. 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 [9, 14]. 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 [14, 38]. Some approaches use existing specification languages [21, 23], and others have their own (specialized) languages for the description of test cases and test execution [9, 14, 38, 40]. All of these works are complementary 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 have 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. There are two complications. The first 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 meaningless test cases, while the second indicates a test failure. The second is that the unit testing framework needs to distinguish three possible outcomes for test cases: a test execution can either be a success, a failure, or it can be meaningless.

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 [15].

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 [28]. 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 and examples that are shipped with JML, 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 `http://www.jmlspecs.org`.

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. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.
8. 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.
9. 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.

10. Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from archives.cs.iastate.edu.
11. Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.
12. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.
13. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, April 2003. Available from archives.cs.iastate.edu.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
21. 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.
22. 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.
23. Pankaj Jalote. Specification and testing of abstract data types. *Computing Languages*, 17(1):75–82, 1992.
24. JUnit. Http://www.junit.org.
25. Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In Pierre Cointe, editor, *Meta-Level Archi-*

tectures 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.

26. 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.
27. 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.
28. 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.
29. 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.jmlspecs.org`.
30. 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.
31. B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.
32. Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, NY, 1992.
33. Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition, 1997.
34. D.J. Panzl. Automatic software test driver. *IEEE Computer*, pages 44–50, April 1978.
35. 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.
36. Reinhold Plosch and Josef Pichler. Contracts: From analysis to C++ implementation. In *Proceedings of TOOLS 30*, pages 248–257. IEEE Computer Society, 1999.
37. Sara Porat and Paul Fertig. Class assertions in C++. *Journal of Object-Oriented Programming*, 8(2):30–37, May 1995.
38. 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.
39. David R. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
40. 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.
41. 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.

42. David Welch and Scott Strong. An exception-based assertion mechanism for C++. *Journal of Object-Oriented Programming*, 11(4):50–60, July/August 1998.