Automated Random Testing to Detect Specification-Code Inconsistencies

Yoonsik Cheon

TR #07-07
February 2007; revised March 2007

Keywords: random testing, test data generator, test oracle, pre and postconditions, JML language.

1998 CR Categories: D.2.5 [*Software Engineering*] Testing and Debugging — testing tools (e.g., data generators, coverage testing); 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.

Published in *Proceedings of the 2007 International Conference on Software Engineering Theory and Practice*, Orlando, FL, July 9–12, 2007, pages 112–119.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Automated Random Testing to Detect Specification-Code Inconsistencies

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, TX 79968-0518
Email: cheon@cs.utep.edu

***Abstract*—An interface specification language such as JML provides a means to document precisely the behavior of program modules such as Java classes, and it is being adopted by industry. However, few practical tools exist for programmers to assure the correctness of their interface specifications. Nonetheless, the correctness of an interface specification is a prerequisite for the use of the specification, both as a precise API documentation and as a foundation for formal verification of and reasoning about the implementation. We propose automated random testing as a practical tool to assure the correctness of interface specifications. The key idea of our approach is to fully automate dynamic, random testing to detect as many inconsistencies as possible between the specification and its implementation. For this, we use a runtime assertion checker as a test oracle, and the goal of our testing is to generate as many non-duplicate test cases as possible that incur a certain type of runtime assertion violations. Our approach has been implemented for Java/JML in a prototype tool called JET, and a preliminary experiment shows that it has a potential to be a valuable testing tool for Java/JML. Our approach can be adapted for other interface specification languages.**

I. Introduction

A formal behavioral interface specification language (BISL) such as JML [1] is becoming popular and being adopted by programmers as a practical tool for documenting and assuring the correctness of programs. This may be partly due to the notational familiarity with and the runtime assertion checking of BISLs. Assertions in a BISL are typically written as boolean expressions of the underlying programming language, often with some extensions such as quantifiers, and such assertions can be executed and thus checked at runtime. However, there are few practical tools available for programmers to assure the correctness of assertions themselves, or that allow the programmers to have more confidence on the correctness of their assertions. For example, the most commonly used JML tools [2] are the type checker (`jml`) and the compiler (`jmlc`) [3], and they provide a limited help assuring the correctness of JML specifications. The former detects only syntax and type errors in specifications, and the latter translates JML specifications into runtime assertion checking code. Although there are also several heavyweight tools for JML (e.g., proof assistants [4], model checkers [5], and symbolic animators [6]), these are not the kind of tools that Java programmers use on a daily basis, and the underlying technology is not mature enough for daily programming uses.

We propose automated random testing as a cost-effective alternative for assuring the correctness of interface specifications and assertions. Testing in general is costly, laborious, time consuming, and error-prone. However, if fully automated, random testing can be an effective tool to detect inconsistencies between a specification and its implementation, as it eliminates the subjectiveness in constructing test cases and increases the variety of input values. Thus, it has a potential for finding errors that are difficult to find in other ways. In addition, automation can reduce the cost of testing dramatically.

The key idea of our approach is to perform dynamic testing by generating test cases randomly and using specifications as test oracles. We test each public method of a class separately. Thus, our test case consists of an optional receiver object and a list of arguments; the receiver is an instance of the class under test, and an argument is either a primitive value or an object. For an argument of a primitive type, we select an arbitrary value of that type randomly. For a class type, we construct a new instance by invoking a constructor and mutate it by invoking a sequence of mutation methods. The constructor and the mutation methods are selected randomly. For this, we classify constructors and methods of a class into several categories (i.e., basic constructor, extended constructor, mutator, and observer) based on their signatures and JML specifications (see Section III-A). We perform a test execution by invoking the method under test with the generated test case. The method is run with runtime assertion checks turned on so that we can observe runtime assertion violation errors. If the execution results in a pre-state assertion violation error such as a precondition error, the test case is invalid because pre-state assertions are the client's obligation. The test objective is to find as many non-redundant test cases as possible that result in a post-state assertion violation error such as a postcondition error; such an assertion violation error means an inconsistency between the specification and its code, as post-state assertions are the implementor's obligations. We use postcondition coverage to identify duplicate test cases.

We implemented our approach for a prototype tool, called JET. We also extended the JML compiler (`jmlc`) and integrated it with JET to collect test coverage information from each test execution. A key implementation feature of JET

is separation of the test case generation module from the rest of the system through a well-defined interface by using several design patterns such as the Observer pattern. We hope this will facilitate a future improvement or experimentation on different test case generation strategies. A preliminary experiment with JET showed a promising result. In a mutation testing experiment, for example, JET detected about 90% of seeded specification faults; the remaining faults are those that weaken the specification, and thus are automatically satisfied by the code. We also found that JET could be a valuable tool for performing regression testing of inherited methods (see Section V).

The rest of this paper is organized as follows. In Section II we briefly introduce JML with an example to be used throughout this paper. In Section III we explain our approach in detail, including automatic generation of test cases, test execution, and test coverage. In Section IV we describe a prototype tool JET that implements our approach. We describe its key features, representation of test cases, and our extension to the JML compiler. In Section V we summarizes our preliminary experimental results on our approach and JET. In Section VI we discuss related work, which is followed by a concluding remark in Section VII.

II. The JML Language and Tools

The Java Modeling Language (JML) is an interface specification language for Java to formally specify the behavior of Java program modules such as classes and interfaces [1], [7], [8]. An interface specification describes formally a program module by specifying both the syntactic interface and the behavior of the module. In JML, the behavior of a program module is specified, for example, by writing pre and postconditions of the methods exported by the module. The pre and postconditions are viewed as a contract between the client and the implementor of the module. The client must guarantee, before calling a method m exported by the module, that m's precondition holds, and the implementor must guarantee that m's postcondition holds after such a call. The assertions in pre and postconditions are usually written in a form that can be compiled, so that violations of the contract can be detected at runtime. Checking pre and postconditions at runtime first pioneered by design-by-contract tools [9] is useful for checking the correctness of a program with respect to its specification.

Fig. 1 shows an example JML specification for a class `Account`, an abstraction of bank accounts. Syntactically, JML assertions are written as special annotation comments in Java source code, either after `//@` (as in line 3) or between `/*@` and `@*/` (as in lines 5–7). As comments, they are ignored by Java compilers but can be used by tools that support JML. Within annotation comments, JML extends the Java syntax with several keywords, such as **`spec_public`**, **`invariant`**, **`requires`**, **`assignable`**, **`ensures`**, and **`pure`**. It also extends Java's expression syntax with several operators such as `\old` and `\result`.

```
1  public class Account {
2    private /*@ spec_public @*/ int bal;
3    //@ public invariant bal >= 0;
4
5    /*@ requires amt >= 0;
6      @ assignable bal;
7      @ ensures bal == amt; @*/
8    public Account(int amt) {
9      bal = amt;
10   }
11
12   /*@ assignable bal;
13     @ ensures bal == acc.bal; @*/
14   public Account(Account acc) {
15     bal = acc.balance();
16   }
17
18   /*@ requires amt > 0 && amt <= acc.balance();
19     @ assignable bal, acc.bal;
20     @ ensures bal == \old(bal) + amt
21     @   && acc.bal == \old(acc.bal - amt); @*/
22   public void transfer(int amt, Account acc) {
23     acc.withdraw(amt);
24     deposit(amt);
25   }
26
27   /*@ requires amt > 0 && amt <= bal;
28     @ assignable bal;
29     @ ensures bal == \old(bal) - amt; @*/
30   public void withdraw(int amt) {
31     bal -= amt;
32   }
33
34   /*@ requires amt > 0;
35     @ assignable bal;
36     @ ensures bal == \old(bal) + amt; @*/
37   public void deposit(int amt) {
38     bal += amt;
39   }
40
41   //@ ensures \result == bal;
42   public /*@ pure @*/ int balance() {
43     return bal;
44   }
45 }
```

Fig. 1. Example JML specification

The annotation in line 2 states that the private field `bal` is treated as public for specification purpose; e.g., it can be used in the specifications of public methods. The most common constraints used in JML specification are preconditions, postconditions, and invariants, specified as **`requires`**, **`ensures`**, and **`invariant`** clauses, respectively. These constraints are all defined using boolean expressions. For example, the class invariant in line 3 states that the balance of an account is always non-negative. As shown in lines 5–7, a method (or constructor) specification precedes the declaration of the method (or constructor). The precondition, specified in the **`requires`**, states that the argument (`amt`) should be non-negative. The frame condition in the **`assignable`** clause of line 6 specifies that the method can assign only to the field `bal`. The postcondition, specified in the subsequent **`ensures`** clause, states that the initial balance of the account, `bal`, should be the same as `amt`. The keyword `\old` in line 20 denotes the pre-state value of its expression; it is most commonly used in the specification of a mutation method such as `transfer` that changes the state of an object. The keyword **`pure`** in line 42 states that the declared method has no side-

effect and thus can be used in JML assertions (as in line 18).

There are various tools that are currently available for JML [2]. For example, the common JML tools, available from the JML website (www.jmlspecs.org), includes a type checker, a runtime assertion checking compiler, runtime support for checking assertions, a unit testing tool, a documentation generator, and a skeleton specification generator. The most interesting of these tools is the JML compiler (`jmlc`) that compiles JML assertions into runtime assertion checking code [3], as it allows us to use JML specifications as test oracles [10]. Method preconditions and postconditions are checked before and after method calls, respectively, and class invariants are checked both before and after method calls. The compilation is transparent in that the behavior of the compiled code is unchanged, except for time and space measure, unless an assertion is violated.

III. Automated Random Testing

Program testing consists of three major steps: test data selection, test execution, and test result determination. The last step called a *test oracle* determines a test success or failure by usually comparing the result of a test execution with the expected result. Our automation approach is to fully automate each step of testing with the goal of finding test cases that reveal inconsistencies between the specifications and code. We test each method of a class separately. For each method of a class, we first generate a test case randomly and then run the method under test by supplying the generated test data; the method is run with runtime assertion checks enabled. If the run results in an assertion violation error, it generally means that there is an inconsistency between the specification and the code; i.e., one or both is incorrect [10]. Otherwise, it means that the code is correct with respect to the specification for that particular test data. Below we explain in detail the key ingredients of our approach—random selection of test data, dynamic testing, and specifications as test oracles.

A. Test Data Generation

We generate test data randomly for each method of the class under test, say C. For a non-static method, a test case is a tuple of objects and values, $\langle r, a_1, \ldots, a_n \rangle$, where r is a receiver object of class C and a_i's are arguments. The type of an argument a_i is either a primitive type, a class, an interface, or an array type. For a static method and a constructor, a test case consists of only arguments, $\langle a_1, \ldots, a_n \rangle$. We use different strategies to generate each element of a test case based on its type. For a primitive type, an arbitrary value of that type is selected randomly; e.g., for an `int` argument, a value from `Integer.MIN_VALUE` to `Integer.MAX_VALUE` is chosen randomly. For an array type, first the dimension is chosen randomly and then the elements are generated one by one by applying the strategy of the element type. In the remainder of this subsection, we explain how we generate a random object of a concrete class; for an abstract class or an interface, null is the only possible value unless one specifies at least one concrete subclass or implementation class in which case a concrete class is randomly chosen to be substituted for the abstract class or the interface.

Because an object's state is hidden, we cannot create an object of an arbitrary state directly. We should do it indirectly by first instantiating a class and changing the instance's state through a series of method calls. Let C be a concrete class with a set of concrete constructors, c_i's, and concrete methods, m_i's; each method m_i is either declared in C or inherited from its superclasses. We classify these constructors and methods into four categories based on their signatures:[1]

1) *Basic constructor*. A constructor c_i is a basic constructor if it doesn't have C as a parameter type. A static method m_i is a basic constructor if its return type is C and it doesn't have C as a parameter type. For example, the constructor `Account(int)` of class `Account` is a basic constructor.
2) *Extended constructor*. A constructor c_i is an extended constructor if it has one or more parameters of type C. A non-static method m_i is an extended constructor if its return type is C and it has one or more parameters of type C. For example, the constructor `Account(Account)` of class `Account` is an extended constructor.
3) *Mutator*. A non-static method is a mutator if its return type is `void`.[2] For example, there are three mutators in the `Account` class: `transfer(int,Account)`, `withdraw(int)`, and `deposit(int)`.
4) *Observer*. All other methods are observers. For example, the `balance()` method of the `Account` class is an observer.

A basic constructor creates a new instance of a class without needing another instance of the same class, whereas an extended constructor can create a new instance of a class only if it is given other instances of the same class. A mutator changes the state of an existing object.

We represent an instance of C as a sequence of constructor and method invocations, $\langle s_0, s_1, \ldots, s_n \rangle$, where s_0 is a basic constructor invocation and the rest are extended constructor invocations or mutator invocations. That is, to generate an instance of a class we first invoke one of its basic constructors and then, to change the instance's state, invoke several extended constructors or mutators. Of course, the constructors and mutators are selected randomly. For example, below are three example invocation sequences for the `Account` class.

1: ⟨new Account(10)⟩
2: ⟨mew Account(10), deposit(10)⟩
3: ⟨new Account(10), transfer(x), withdraw(50)⟩

[1]JML specifications can be used to make a more accurate classification. For example, a pure method is an observer, a method with an **`assignable`** clause (stating frame conditions) may be a mutator, and the JML assertion `\`**`fresh`**`(x)` (stating that `x` is newly created) can be used to determine whether a method is a constructor or not. However, this is not yet implemented in JET (see Section IV).

[2]In reality, many mutation methods have return types other than `void`; e.g., `Collection.add` returns true if the collection was changed as the result of the call. In JET, the classification of methods and constructors can be overriden by the user.

where x is $\langle$new Account(20)$\rangle$

As shown in the last example, creating an instance may require creating a whole bunch of new objects, i.e., argument objects, arguments of the arguments, etc. In fact, an instance $\langle s_0, s_1, \ldots, s_n \rangle$ denotes a collection of object graphs, and each object in the collection of the object graphs has to satisfy its class invariants. In addition, each invocation has to meet the precondition of the method or constructor being invoked. Therefore, not all instances—represented as invocation sequences—are feasible. For example, the last sequence fails to create an instance because the `withdraw` call fails to meet its precondition; the account has an insufficient balance (30) for the withdrawal request (50).

B. Test Execution

Our approach is dynamic in that we run the method under test to generate its test cases. We run the method for two purposes: to filter out generated test cases and to determine the test result. For both, we use JML specifications as a decision procedure. A generated test case may be inappropriate for testing the method. For example, the test case may not satisfy the precondition of the method. As the precondition is the client's obligation, such a test case is an invalid input to the method and is referred to as *meaningless* [10]. For example, a test case $\langle\langle$new Account(10)$\rangle$, 20$\rangle$ is meaningless for the `withdraw` method, though it's meaningful for the `deposit` method.

A test case is *redundant* if an equivalent test case already exists in the set of generated test cases, referred to as a *test suite*. If $\langle\langle$new Account(20)$\rangle$, 30$\rangle$ is in the set, then $\langle\langle$new Account(10), deposit(10)$\rangle$, 30$\rangle$ is redundant because the two receivers have the same state. The redundancy of test cases depends on the (observable) equivalence of objects [11]. However, checking the equivalence of objects are often very expensive, so in practice an approximation is used, often based on the notion of test or code coverage. In branch coverage, for example, if a test case covers the same branch as already covered by another test case, the test case is redundant. As our testing goal is to detect inconsistencies between code and its specification, we define the redundancy of test cases in terms of specifications (see Section III-C).

As in [10], we use JML specifications as test oracles. A post-state assertion such as a method postcondition and a class invariant is used as a test oracle. The method under test is executed with the generated test case. If the execution results in a postcondition violation error or a post-state invariant violation error, the test fails because it means that the method doesn't satisfy its specification for that particular test data; i.e., there is an inconsistency between the code and its specification.

C. Test Coverage

In structural testing (a.k.a. white box testing), test coverage or more accurately code coverage refers to the degree of code that is exercised by a set of test cases. Thus, it provides an indirect measure of quality of a set of test cases, and also identifies redundant test cases. As the primary goal of our testing is to find inconsistencies between code and its specification, we use test coverage based on postconditions, and there are several possibilities. For example, we can view the whole postcondition as a single boolean expression to falsify with a test case, consider each boolean sub-expression separated by logical operators, or even consider every possible combination of boolean sub-expressions; in structural testing, these are called decision coverage, condition coverage, and multiple condition coverage, respectively. Another alternative is to transform the postcondition into some normal form (e.g., a conjunctive normal form $a_1 \wedge a_2 \wedge \ldots \wedge a_n$) and consider each clause separately [12]. In JET, we implemented multiple condition coverage (see Section IV). That is, the tool tries to find as many test cases as possible with different combinations of boolean sub-expression values that falsify the whole postcondition.

IV. Tool Support

We developed a prototype tool called JET[3] that supports our approach of automatically detecting inconsistencies between code and its specification (see Fig. 2 for a sample screen of JET). JET also provides a simple integrated development environment for Java/JML to edit a source code file with a syntax-highlighted editor, to compile it with a Java or JML compiler, and to run the compiled bytecode class. A typical use of JET is fist to compile Java/JML source code with the built-in JML compiler and then to drop the bytecode file to or to load it from the tester. The tester shows all public constructors and methods of the loaded class. One can tests different sets of the methods (e.g., all methods, inherited methods, non-inherited methods, and explicitly-selected methods) by using the mouse and the pop-up test menu. JET tests each method from the selected set of methods, one at a time, by showing generated test with the test results as they are being generated and tested. By selecting and clicking a particular test case, one can see the detailed information of the test case, such as the values and, for a failed test case, a runtime assertion violation error message that explains the reason of the failure. If one clicks the error message, the built-in editor highlights the violated JML assertion so as to facilitate tracking of the cause of the test failure.

A. Architecture

JET consists of several components or modules, implemented as separate Java packages, e.g., editor, tester, and JML compiler. The tester is responsible for performing automated testing by generating test cases, executing them, and deciding test results. It can also export generated test cases so that they can be used for debugging or regression testing (see Section IV-C). A distinguishing feature of our implementation is that the logic of test case generation and its implementation are completely decoupled and separated from the rest of the system through well-defined interfaces. The Observer design

[3]JET stands for Evolutionary Testing for Java. Although it currently supports only random testing, our long-term goal is to apply evolutionary techniques (e.g., genetic algorithms) to testing Java programs.

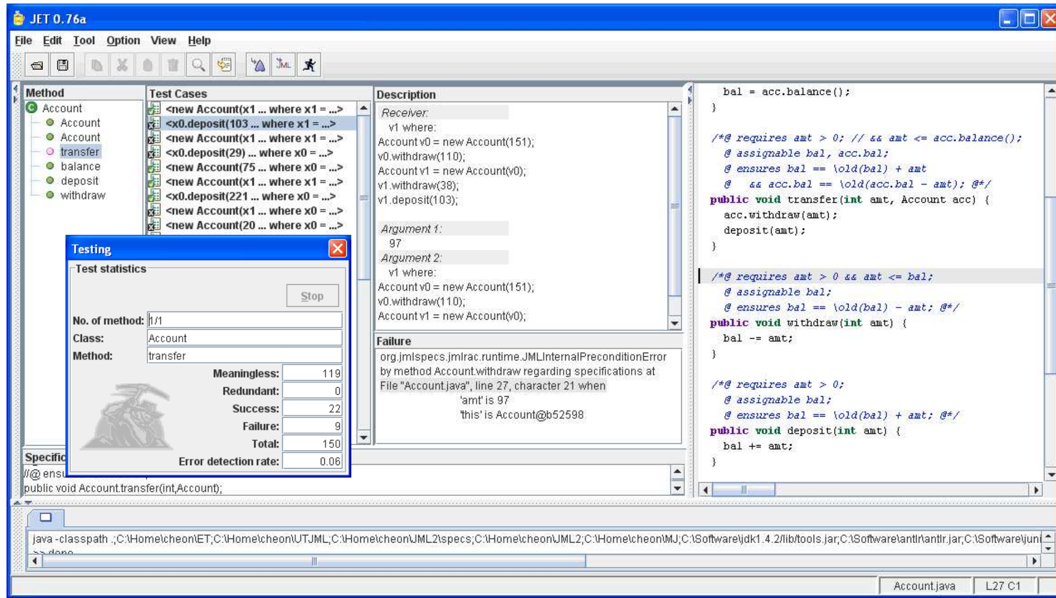


Fig. 2. Sample screenshot of JET

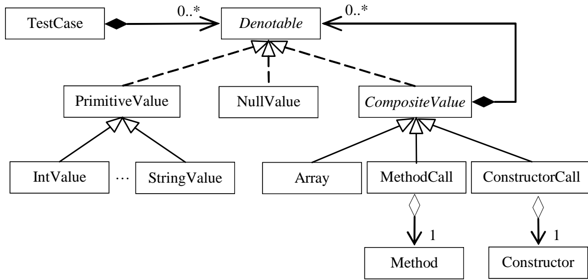


Fig. 3. Representation of Test Cases

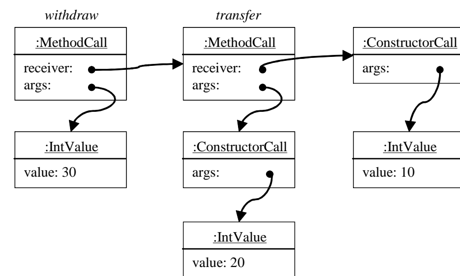
pattern was used for this, and the test case generator can be run even from the command-line by providing a simple console-based observer. This separation makes the test case generator pluggable, and we hope it will facilitate our future improvements and experimentations on different test case generation techniques (e.g., an evolutionary approach).

B. Representation of Test Cases

Conceptually, a test case is a tuple of objects and values, $\langle r, a_1, \ldots, a_n \rangle$, where r is an optional receiver and a_i's are arguments. Test cases should be represented in such a way as to facilitate both dynamic (runtime) and static (compile-time) constructions. This is because we not only construct them dynamically at runtime but also export them for a later test execution, e.g., for regression testing (see Section IV-C).

Our representation of test cases is shown in Fig. 3. A test case is composed of a number of denotables, i.e., a receiver and arguments. The interface `Denotable` provides a common interface to different types of Java values. It declares such methods as `value` that returns the represented Java value or object, `code` that returns Java source code that, if evaluated, gives the represented value, and `type` that returns the type of the represented value. A non-array object is represented as either a method call or a constructor call. The `value` methods of these classes invoke the corresponding Java method or constructor by using Java's reflection facility to build the represented value.

We use the Composite design pattern to represent an object, i.e., a sequence of method and constructor calls. This is necessary because the receiver or arguments of a call may be objects (e.g., another constructor or method call); in fact, an object in our representation really denotes an object tree or graph. For example, an object $\langle$new Account(10), transfer(x), withdraw(50)$\rangle$, where x is $\langle$new Account(20)$\rangle$, is represented as:



C. Exportation of Test Cases

JET allows one to export generated test cases as JUnit test classes; e.g., one can export a selected set of test cases, all failed test cases, or all the test cases. This is to aid debugging both the code and its specification, once an inconsistency between them is detected, and also to support regression testing. JUnit is a popular unit testing framework for Java to assist in organizing and executing a large number of tests [13]. One can also add hand-written test data to the generated JUnit test class, preferably as a subclass, to take advantages

```
 1  public void oracleDeposit(Account receiver,
 2                            int arg0) {
 3    try {
 4      receiver.deposit(arg0);
 5    }
 6    catch (JMLEntryPreconditionError e) {}
 7    catch (JMLAssertionError e) {
 8      String msg = /* compose err msg */;
 9      fail(msg);
10    }
11    catch (Exception e) {}
12  }
13
14  public void testDeposit() {
15    Account receiver = null;
16    int arg0 = 0;
17    receiver = new Account(new Account(77));
18    arg0 = 158;
19    oracleDeposit(receiver, arg0);
20  }
```

Fig. 4. Sample test oracle and data methods

of two complementary techniques of automated and manual testing.

The generated JUnit test class contains several types of methods. In addition to several boiler-plate JUnit methods such as a test suite method [13], the test class contains two types of methods that are specific to JET: *test oracle methods* and *test data methods* (see Fig. 4). A test oracle method is responsible for deciding a test result; this is done by detecting any occurrence of assertion violation errors during a test execution [10]. One test oracle method is generated for each method whose test cases are exported. A test data method is a JUnit test method and responsible for performing a test execution by first building the test data and calling an appropriate test oracle method. For each exported test case, a separate test data method is generated; i.e., each test case becomes a separate JUnit test method so that JUnit can properly report a test summary (e.g., number of failures).

D. Extension to the JML Compiler

We extended the JML compiler (`jmlc`) [3] and integrated it to JET. The extension was made to obtain test coverage information from the runtime assertion checking code generated by the compiler.[4] Whenever a postcondition is evaluated, information about which sub-expressions are evaluated to true, which are evaluated to false, and which are not evaluated at all (due to JML's short-circuit evaluation) is collected and recorded. JET retrieves this information to determine the redundancy of test cases and also to calculate test coverage. We used the Visitor design pattern to make this extension.

V. Evaluation

We performed a simple experiment with JET to evaluate both the tool itself and our approach in general. Although the tool is still under development, our experiment showed a promising result and also suggested several improvements.

[4]The extension also include a new JML language construct, called a *call sequence clause*, to specify the allowed sequences of method calls in a clear and concise notation [14].

The JML distribution available from the JML website (www.jmlspecs.org) contains several example specifications with sample implementations. One such an example is the package `org.jmlspecs.samples.digraph` that contains implementations of two different types of directed graphs with fairly complete specifications. There are total nine classes in the package, including one interface and two abstract classes, with 1099 lines of source code, including comments and JML specifications. We ran JET on these classes. For the interface and two abstract classes, we specified all their concrete implementation classes or subclasses so that the tool can use instances of these classes in places where objects of the interface or abstract classes are required, e.g., arguments of method invocations. If multiple concrete classes are specified for the same interface or abstract class, the tool picks one randomly. To our disappointment, JET was not able to find any inconsistencies for these classes. Perhaps, the code and its specifications are debugged and tested thoroughly. In fact, the package is shipped with manually written test cases for the JML-JUnit unit testing tool [10] that, like JET, uses the runtime assertion checker as test oracles.

We next performed a mutation testing experiment to evaluate the effectiveness of our approach. Mutation testing is based upon seeding a fault to a program and determining whether testing identifies the seeded fault. If a test case distinguishes between the mutated program (referred to as *mutant*) and the original program, it is said to *kill* the mutant. In our experiment, we mutated the specification, not the code, by introducing three mutation operations: value replacement that replaces a value with another value of the same type (e.g., true for false), operator replacement that replaces an operator with another, and variable replacement that replaces a variable with another variable or value of the same type. We seeded total 30 faults manually[5], and JET killed 22 mutants, giving a 73% kill rate. We examined each of the surviving mutants, and noticed that out of eight surviving mutants, five mutants couldn't be killed because of other seeded faults; e.g., a fault seeded in `setPredecessor` prevented JET from creating an instance of class `SearchableNode` that has a predecessor, thus a fault seeded in `getPrececessor` couldn't be detected. Once we removed the interfering faults, JET killed all these five surviving mutants, giving a 90% kill rate. The remaining three surviving mutants are similar to what are referred to as equivalent mutants, mutants that have the same behavior as the original program. In our case, these are often weakened specifications, e.g., the specification of the `clone` method of class `Arc` mutated to: `\result instanceof Arc ==> ((Arc)result).equals(this)`, where the original specification has `&&` instead of `==>`. This is a fundamental problem of an assertion-based approach in that a missing assertion can't be checked and thus detected.

We found that JET has the potential to be an excellent testing tool for Java and JML. For example, it greatly simplifies

[5]We tested class invariants and constructor specifications as separate runs, as faults in them prevented JET from creating any objects.

regression testing of inherited methods in inheriting contexts (i.e., subclasses), which is one of the most laborious and time consuming tasks and often not done at all when a new subclass is introduced.[6] In JET, it requires just one button click. JET can also be a great help in preventing divergence between an implementation and its documentation, a major problem in program maintenance.

However, we also noticed several shortcomings of our approach. The main shortcoming is that when a method has a complex precondition or require several objects as arguments, the majority of generated test cases become meaningless; it doesn't satisfy the precondition, or the receiver or an argument object doesn't satisfy its class invariants. For example, for the `transfer` method of the `Account` class, more than 98 percents of the generated test cases became meaningless. In summary, it is unlikely to be able to build successfully an object graph consisting of many objects, each object satisfying its class invariant, by randomly picking up constructors and mutation methods.

A short-term solution that we are working on now is to build objects incrementally, by checking a constructor or method call one at a time on-the-fly when it is chosen, and to share the successfully-built objects. Currently, all the call sequences are determined for the whole object graph before even starting constructing any objects, and there is no sharing of objects for a single test case or among different test cases. The incremental approach will surely require more time to construct an object, as the runtime assertion checking currently is very expensive, but it will be more likely to produce a valid object. However, our long-term research goal is to introduce heuristic or meta-heuristic techniques (e.g., genetic algorithms) to guide the search for valid objects and meaningful test cases [15], [16].

VI. Related Work

There are several tools for automatic generation of object-oriented unit tests, such as JCrasher [17] and Jtest [18], that generate sequences of method invocations. JCrasher tests the robustness of Java programs by causing the program under test to crash, i.e., to throw an uncaught exception. As in our approach, it performs dynamic testing using reflection and generates a (huge) random set of test cases based on the method signatures. However, JCrasher uses no formal specifications to guide testing, e.g, as test oracles, to classify methods, or to prune generated test cases. Jtest [18] is a commercial tool from Parasoft that supports automatic white box testing. It generates a minimal set of test cases based on code analysis, e.g., a set of test cases that execute every possible branch of the method under test. As in JCrasher, however, it doesn't use formal specifications.

The idea of automatically generating test oracles from formal specifications is not new. Peters and Parnas proposed a tool that generates a test oracle from formal program documentation written in tabular expressions [19]. The test oracle procedure, generated in C++, checks if an input and output pair satisfies the relation described by the specification. Cheon and Leavens proposed a novel approach of employing a runtime assertion checker as a test oracle engine [10]. In our approach, we promoted this idea further by dynamically generating test data and feeding it to the oracle to detect inconsistencies between code and its specification.

Many research papers have been published on techniques and tools for runtime checking inconsistencies between specifications and their implementations (e.g., [20], [21], and [22]). Most of the approaches are similar to multi-version programming except that one version is a specification, serving as an oracle. For example, Antoy and Hamlet describes an approach to checking the execution of an abstract data type's implementation against its algebraic specification [20]. The algebraic specification is executed by (term) rewriting, and compared with the execution of the implementation by using a user-provided abstraction function. Barnett and Schulte's work is similar in that specifications, written in AsmL [23], are executed separately from programs, and violations are detected by comparing the two outputs [21]. Unlike these work, our approach is based on design-by-contract and different in that assertions are written in terms of program values and thus there is no separate (specification) program running. Nunes et al. recently proposed an interesting way of checking the conformance of Java classes against algebraic specifications [22]. The essence of their approach is to translate an algebraic specification into JML assertions so that it can be checked at runtime by JML's runtime assertion checker. However, as in Antoy and Hamlet's work, the approach is not fully automated, as one has to provide a mapping between Java classes and algebraic specifications and supply test data to execute the translated code and assertions. Our approach is complementary and can be used to supply test data to the translated code and JML assertions to test them automatically.

One problem of generating test cases randomly for object-oriented program is the redundancy of test cases. As the state of an object is indirectly represented as a sequence of method invocations (see Section IV-B), it is hard to decide whether two method invocation sequences lead to an equivalent object state or not. Xie, Marinov, and Notkin proposes a framework for detecting redundant test cases [11]. The framework supports several different techniques for detecting equivalent object states, e.g., comparing the whole or parts of call sequences, comparing concrete states, and using the `equals` method. To select, from a large set of test cases, a small subset likely to reveal faults, Pacheco and Ernst compare the program's behavior on a given test case against an operational model of correct operation, derived from an example program execution [24]. Our approach to this problem to use a postcondition-based coverage, and it is simple and efficient to implement; if two test cases violate the same sub-expression of a postcondition, one is redundant.

[6] In object-oriented programs, every inherited method may need to be retested in subclasses because an inherited method may call directly or indirectly a method that is overridden by the subclasses.

VII. CONCLUSION

We proposed automated random testing as a practical tool for assuring the correctness of formal interface specifications. Our contribution is not a novel new idea or technique but engineering a practical solution by combining and integrating existing ideas and techniques. We took the idea of employing a runtime assertion checker as a test oracle, and combined it with the technique of dynamic, random testing to detect inconsistencies between code and its specification. We implemented the approach in a prototype tool for Java/JML, called JET. The most distinguishing feature of JET is full automation of unit testing, from test data generation to test execution and test result determination; e.g., with one click of button, one can find inconsistencies between Java code and its JML specifications. A preliminary experiment showed a promising result in that JET can detect many specification and code errors, especially for methods with simple preconditions and invariants. However, there are several places that can be further improved in the future. One such an improvement would be to introduce heuristic or meta-heuristic techniques (e.g., genetic algorithms) to guide the search for test data that satisfy complex method preconditions and class invariants.

ACKNOWLEDGMENT

The work of the author was supported in part by the NSF, under grant CNS-0509299. The author thanks the anonymous reviewers for their comments and suggestions. Thanks to Perla Escarcega for writing the initial code for the test case exporter of JET.

REFERENCES

[1] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, June 2005.

[3] Y. Cheon, "A runtime assertion checker for the Java Modeling Language," Department of Computer Science, Iowa State University, Ames, IA, Tech. Rep. 03-09, Apr. 2003, the author's Ph.D. dissertation.

[4] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS 4334. Springer-Verlag, 2007.

[5] Robby, E. Rodriguez, M. B. Dwyer, and J. Hatcliff, "Checking JML specifications using an extensible software model checking framework," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 280–299, 2006.

[6] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting, "Symbolic animation of JML specifications," in *Proceedings of the International Conference on Formal Methods 2005 (FM'05)*, ser. Lecture Notes in Computer Science, vol. 3582. New York, NY: Springer-Verlag, July 2005, pp. 75–90.

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

[8] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185–208, Mar. 2005.

[9] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[10] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, ser. Lecture Notes in Computer Science, B. Magnusson, Ed., vol. 2374. Berlin: Springer-Verlag, June 2002, pp. 231–255.

[11] T. Xie, D. Marinov, and D. Notkin¡, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering (ASE'04)*. IEEE, 2004, pp. 196–205.

[12] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353–363, May 1994.

[13] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 37–50, 1998.

[14] Y. Cheon and A. Perumendla, "Specifying and checking method call sequences of Java programs," *Software Quality Journal*, vol. 15, no. 1, pp. 7–25, Mar. 2007.

[15] Y. Cheon, M. Kim, and A. Perumendla, "A complete automation of unit testing for Java programs," in *International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, June 27-29, 2005*, 2005, pp. 290–295.

[16] Y. Cheon and M. Kim, "A fitness function for evolutionary testing of object-oriented programs," in *Genetic and Evolutionary Computation Conference, Seattle, WA, USA, July 8-12, 2006*. ACM Press, July 2006, pp. 1952–1954.

[17] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software—Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, Sept. 2004.

[18] Parasoft Corporation, "Automatic Java software and component testing: Using Jtest to automate unit testing and coding standard enforcement," available from urlhttp://www.parasoft.com, as of Jan. 2007.

[19] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, Mar. 1998.

[20] S. Antoy and D. Hamlet, "Automatically checking an implementation against its formal specification," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 55–69, Jan. 2000.

[21] M. Barnett and W. Schulte, "Runtime verification of .NET contracts," *The Journal of Systems and Software*, vol. 65, no. 3, pp. 199–208, Mar. 2003.

[22] I. Nunes, A. Lopes, V. Vasconcelos, J. ao Abreu, , and L. S. Reis, "Checking the conformance of java classes against algebraic specifications," in *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, ser. Lecture Notes in Computer Science, Z. Liu and H. Jifeng, Eds., vol. 4260. New York, NY: Springer-Verlag, Nov. 2006, pp. 494–513.

[23] M. Barnett and W. Schulte, "The ABCs of specification: AsmL, behavior, and components," *Informatica*, vol. 25, no. 4, pp. 517–526, Nov. 2001.

[24] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 — European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, A. Black, Ed., vol. 3586. Berlin: Springer-Verlag, 2005, pp. 504–527.