JMLUnit: The Next Generation

Daniel M. Zimmerman and Rinkesh Nagmoti

Institute of Technology
University of Washington Tacoma
Tacoma, Washington 98402, USA
dmz@acm.org, rinkeshn@u.washington.edu

Abstract. Designing unit test suites for object-oriented systems is a painstaking, repetitive, and error-prone task, and significant research has been devoted to the automatic generation of test suites. One method for generating unit tests is to use formal class and method specifications as test oracles and automatically run them with developer-provided data values; for Java code with formal specifications written in the Java Modeling Language, this method is embodied in the JMLUnit tool and the JUnit testing framework on which it is based. While JMLUnit can provide reasonable test coverage when used by a skilled developer, it suffers from several shortcomings including excessive memory utilization during testing and the need to manually write significant amounts of code to generate non-primitive test data objects. In this paper we describe JMLUnitNG, a TestNG-based successor to JMLUnit that can automatically generate and execute millions of tests, using supplied test data of only primitive types, without consuming excessive amounts of memory. We also present a comparison of test coverage between JMLUnitNG and the original JMLUnit.

1 Introduction

Unit testing has been an important validation technique in software development processes for many years. In a typical unit testing process, a developer designs a set (or *suite*) of unit tests and runs them on the system under test (SUT). Each individual unit test is designed to demonstrate that some subset of the software (the *unit* being tested) performs appropriate actions and generates appropriate outputs given particular inputs and a particular starting state. The existence of a comprehensive unit test suite provides evidence for the stability, reliability, and security of the system, though it cannot guarantee the system's correctness.

Unfortunately, designing test suites is a painstaking, repetitive, and error-prone task, especially for large, complex software systems. Test developers can easily overlook critical situations that need testing or develop a test suite with poor *coverage*—that is, one that tests an insufficient fraction of a system's code or functionality. Moreover, the manual development and maintenance of test suites (regardless of quality) represents a significant portion of the development and maintenance costs for a complex software project.

To address both the coverage and cost issues, there has been significant research effort devoted to the automatic generation of high-coverage unit test suites using techniques ranging from purely random test generation to the use of symbolic execution to find critical execution paths. While some of these techniques can provide reasonable test coverage at low cost, they all have various limitations and have seen little adoption by software developers.

This work focuses on improving one particular unit test generation technique that has been adopted by developers who use the Java Modeling Language (JML) to specify their software systems, namely the specification-based test generation embodied in the JMLUnit tool and the JUnit testing framework on which it is based. After providing some background information about unit testing, JML, and JMLUnit, we describe the limitations of JMLUnit for testing complex systems. We then address these limitations with JMLUnitNG, a successor to JMLUnit based on the TestNG testing framework. Finally, we demonstrate our improvements using coverage results from tests generated by both JMLUnit and JMLUnitNG. The goals of this work are to make automated unit test generation for JML-annotated Java programs more effective and easier for developers and, more importantly, to provide a platform upon which to conduct experiments with new test data generation techniques that are currently under development.

2 Background

2.1 Unit Testing

Unit testing is, essentially, the execution of individual components of a system (the *units*) in specific contexts to see whether they generate expected results. A single unit test has two main parts: the *test data*, which are the actual values for software entities such as method parameters that will be used to set up the state of the unit under test, and the *test oracle*, which is a piece of code that determines whether the behavior of the unit is "correct" when it is set up with the test data and executed. A given SUT typically requires many unit tests, which are collectively called a *test suite*. The quality, or *coverage*, of a particular test suite can be measured in several ways [16]; for example, *code coverage* is the percentage of the executable code in the SUT that is actually executed when running the test suite.

The simplest way to create unit tests is to rely on human judgment: a developer sits down with a piece of software, decides what test data should be used and how to determine whether each test has passed or failed, and encodes this information manually. Despite the fact that many techniques for automated test data and test oracle generation have been developed over the last several years, most unit test generation is still done by hand, even in large systems. For example, the open-source Eclipse Development Platform[1] contains several thousand hand-written unit tests.

[1] http://www.eclipse.org/

There are several ways to generate both test data and test oracles automatically. One such way, the focus of this work, is embodied in the JMLUnit tool (described in Section 2.3); we will briefly describe some others in Section 6.

2.2 The Java Modeling Language

The Java Modeling Language (JML) [13] is a specification language for Java programs. It supports class and method contracts in a Design by Contract [14] style, as well as more sophisticated properties up to and including full mathematical models of program behavior. Several tools work with JML, including compilers, static checkers, test generators, and specification generators [6].

The *Common JML* tool suite is the original, and still most widely used, set of JML tools. It supports Java language versions up to 1.4 and includes a type checker (`jml`), a compiler (`jmlc`) that compiles JML annotations into runtime checks, a runtime assertion checker (`jmlrac`), a version of Javadoc (`jmldoc`) that generates documentation including JML specifications, and a unit testing framework (JMLUnit, described below).

Support for modern Java (1.5 and later) syntax in JML—including generic types, enhanced for loops, and annotations—is currently being developed in OpenJML,[2] based on the current OpenJDK[3] codebase, and JMLEclipse,[4] based on the Eclipse Development Platform.

2.3 JMLUnit

JMLUnit [7] is a unit testing framework for JML-annotated code. It takes advantage of JML runtime assertion checking (hereafter, *RAC*) to enable the automatic construction of test oracles that classify tests into three categories: *successful* (or *passed*), *unsuccessful* (or *failed*), and *meaningless*. Successful and unsuccessful tests are familiar concepts to developers experienced in unit testing. In the JMLUnit context, a successful test is one where a method is called and no RAC errors occur; this means that the method conforms to its specification with respect to that call. An unsuccessful test is one where a method is called with its precondition satisfied and a RAC error occurs; this means that the method does not conform to its specification, because once its precondition has been satisfied it must execute correctly without violating any assertions.

Meaningless tests, on the other hand, are not likely to be familiar to most unit testing practitioners. In the context of JMLUnit, a meaningless test is one where a method is called *without* its precondition satisfied, causing a RAC error before the method is executed. In JML (and other Design by Contract-based specification techniques), a method call is explicitly permitted to generate any result whatsoever when it is called without its precondition satisfied, ranging from an unchanged system state to a catastrophic system failure. Since any

[2] http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/OpenJML/
[3] http://openjdk.java.net/
[4] http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/JmlEclipse/

result of such a test must be acceptable by definition, there is no way for such a test to fail; a test that cannot fail gives no useful information and is therefore meaningless.

Of course, test oracles generated from the JML specifications present in the SUT are necessarily limited by the scope of those specifications. Some JML specifications are not executable, so the runtime checker cannot catch all possible specification violations (though the range of violations it can catch is extensive). The more detailed and precise executable specifications exist for a method, the better the ability of the generated test oracles to discern the correctness of that method. Methods or classes with no executable specifications—that is, with only informal specifications or with formal specifications that cannot be checked at runtime—cannot be effectively tested using such test oracles. However, the problem of writing good executable class and method specifications, while extremely important, is beyond the scope of this work; we proceed under the assumption that good executable specifications are present in at least a reasonable fraction of any system we intend to test.

In addition to constructing a test oracle for every method in the SUT, JMLUnit also constructs a limited set of test data for each method. It uses a default set of values for each primitive type in the Java language as well as the `String` type, which it treats as a primitive type for testing purposes. For example, the default set of values for the `int` type is {`-1`, `0`, `1`} and the default set of values for the `String` type is {`null`, `""`} (`""` is the empty string). JMLUnit allows the developer to augment these default sets with additional values; the test code it generates has a clearly delineated "test data supply section" where the developer can specify data values to be used in addition to the defaults. Typically, JMLUnit generates two test classes (one containing the test oracles and one containing the test data) per class under test; however, there is also an option to relegate the test data for all classes under test to a single "test data generator" class. JMLUnit does no automatic test data generation for non-primitive types, relying solely on the developer to write the code that generates such test data.

The tests generated by JMLUnit are executable by JUnit,[5] one of the first and most widely used automated test execution frameworks for Java-based systems. They exhaustively use all combinations of the generated test data as parameters to each method under test. For example, consider method `m` in Figure 1, which takes one `int` parameter and one `String` parameter. JMLUnit has 3 default `int` values and 2 default `String` values, so `m` will be called 6 times during testing if only default values are used. If the default values are augmented with i additional `int` values and s additional `String` values, `m` will be called $(3 + i)(2 + s)$ times.

JMLUnit includes a custom JUnit test runner (`jml-junit`) that provides detailed reporting of test results and correctly handles meaningless tests; JUnit itself has no integrated concept of meaningless tests. The JUnit framework is also integrated into the Eclipse IDE and JMLUnit tests can be run directly from inside Eclipse, though doing so causes meaningless tests to be reported as passed tests and the test results to be reported with less detail.

[5] http://www.junit.org/

```
public class Exemplar {
  public Exemplar(String s0, String s1, String s2, String s3,
                  byte b, char c, Other o, Thing t) {
    // constructor body omitted
  }

  public int m(int one, String two) {
    // method body omitted
  }
}
```

Fig. 1. An exemplar of a Java class skeleton.

3 Shortcomings of JMLUnit

In the hands of a skilled developer, JMLUnit can generate tests with good coverage; however, it has several limitations that make it somewhat impractical to use for large, complex systems. One of these is that it does not attempt to automatically generate non-primitive test data, leaving that task entirely to the developer. This requires the developer to manually write methods that return specific test objects in response to specific requests. In its generated test classes, JMLUnit provides skeletons for these methods, which are intended to return specific test data objects indexed by integers.

Consider class `Exemplar` in Figure 1, which has a constructor with the same signature as one we used in our experiments. When JMLUnit generates tests for the `Exemplar` constructor, it creates a method to provide objects of class `Thing` for the last constructor parameter. The developer must fill in the body of that method so that, whenever JMLUnit requests the `Thing` with index n, the method returns whatever the developer has decided the nth `Thing` should be. In most cases, it is important that the test object be a fresh copy, because the order in which tests are run is not known a priori and reuse of test objects can cause test results to unintentionally depend on the order in which the tests are run. Similarly, it is important that the test objects be constructed deterministically, because otherwise the test results might vary across test runs even if nothing in the SUT has changed. This leads to an implementation style where data generation methods are large `switch` statements, with the developer writing code in each `case` of the `switch` statement to generate a single test object; in fact, the skeleton code generated by JMLUnit is exactly such a `switch` statement with a `default` case that generates no test data. Such code requires considerable developer effort both to write and to maintain.

In addition to requiring data generation methods as above, JMLUnit does not provide a reasonable way to specify distinct test data sets for distinct contexts. For the `Exemplar` above, JMLUnit generates and provides extension points for `String`, `char` and `byte` data sets, as well as providing extension points for the developer to generate data for `Other` and `Thing`; however, it only provides *one* such data set and extension point for each type. Thus, if the 4 `String` parame-

ters `s0` ... `s3` have significantly different requirements (e.g., `s0` must be parsable as a number while `s2` must be a capitalized last name with certain length restrictions), the developer must add test data to the single `String` data set that satisfies all these requirements. This results in many meaningless tests where numeric strings are used as names and vice-versa.

The most critical shortcoming of JMLUnit, however, is its memory utilization. Since it relies on JUnit as its execution engine, JMLUnit must construct an entire JUnit test suite in memory, including all the test data to be used, before a single test is run. As described above, JMLUnit exhaustively tests all combinations of the generated test data for each method under test; thus, a single method that takes multiple parameters can result in extremely large numbers of tests. For the `Exemplar` constructor, if the developer gives no additional values beyond the default sets for the primitive types and `String` and generates 2 test objects for each of the `Other` and `Thing` types, JMLUnit generates a total of 384 tests. However, in a more realistic scenario where the developer adds, e.g., 3 `char` values, 2 `byte` values, and 2 `String` values to the default sets and generates 4 test objects for each of the object types, JMLUnit generates 102,400 tests.

The combinatorial explosion caused by adding additional test values is not problematic in itself; each of those 102,400 tests would execute quite quickly on any modern machine. However, the fact that JMLUnit is forced to construct the entire test suite in memory before executing the tests is a serious problem, because it makes such test suites completely impractical to execute even on extremely capable hardware. We attempted to run such a test suite for a case study (described in Section 5) on our test machine, an Apple Xserve with two 3.0GHz quad-core Xeon processors and 18GB of memory; even allowing the Java virtual machine to use 16GB of heap space, we found that it exhausted available memory before giving the results of a single test.

4 JMLUnitNG: Improvements to JMLUnit

In order to test more complex systems with less developer intervention, we have created a new tool called *JMLUnitNG*. The new tool addresses the shortcomings described in the previous section while preserving most of the basic operating principles of the original JMLUnit.

4.1 Test Data Generation

The first shortcoming we address is the lack of non-primitive test data generation. To test `Exemplar`, we need test data of class `Thing`. `Thing` has at least one constructor, either the default no-argument constructor provided by Java in the absence of any constructor code or an explicit constructor that takes zero or more parameters.

If `Thing` has a default constructor, we can construct `Thing`s by using that default constructor. If `Thing` has explicit constructors, tests will be generated for each of them when we generate tests for class `Thing` itself; thus, construction of a

number of `Thing`s will necessarily be attempted as part of the testing process. We can use the `Thing` constructors and their test data to generate `Thing`s for use as test data in other contexts; if there are k tests generated for `Thing` constructors, that gives us at most k `Thing`s for testing other (non-constructor) methods of `Thing` and methods of other classes under test that take `Thing` parameters. We have at most k instances, rather than exactly k instances, because some of the constructor tests may be meaningless or may fail; such tests do not result in the creation of `Thing`s suitable for further testing.

We use Java reflection to generate these instances. Like JMLUnit, JMLUnitNG generates two classes—one containing test oracles and another containing test data—per class under test. In each test data class, JMLUnitNG creates an inner class that iterates over the instances that are successfully created during constructor tests. When we run JMLUnitNG on class `Exemplar`, which takes a `Thing` as a constructor parameter, JMLUnitNG inserts code into the test data class for `Exemplar` that uses Java reflection to search for the test data class for `Thing`. Later, when running the tests on `Exemplar`, JMLUnitNG can then find the test data class for `Thing` (if it exists on the classpath) and use it to obtain `Thing`s for testing. The developer can also directly specify `Thing`s, as in the original JMLUnit. If JMLUnitNG finds the test data class for `Thing` when the tests are run, and reflective test object generation is enabled, the generated `Thing`s are used in addition to the developer-specified `Thing`s; if not, only the developer-specified `Thing`s are used.

There are three main issues that arise when using reflection and constructor test cases to generate test data. The first issue is that it is possible to have cyclic dependencies; for example, a constructor (not necessarily the only constructor) of class `X` takes a parameter of class `Y` and a constructor (again, not necessarily the only one) of class `Y` takes a parameter of class `X`. This issue can be addressed in a straightforward, though perhaps not optimal, way: use cycle detection flags when instantiating objects, such that if an instance of `X` is requested when another instance of `X` is already in the process of being generated, the cycle is detected and stopped by providing a default (that is, generated by a default constructor) or developer-specified instance of `X` instead of dynamically constructing one from test data.

The second issue is that constructing test data using reflection does not take polymorphism into account. For example, given a method on a chessboard class that takes a `Piece` as a parameter, JMLUnitNG will attempt to generate `Piece` objects but will not attempt to generate, e.g., `Bishop` or `Knight` objects even if those classes extend `Piece` and have test data generators. This issue is difficult to address in the general case, such as when determining what types to generate for a method that takes an `Object` as a parameter. It can be addressed for certain classes, e.g., the Java Collections Framework, with simple test data generation rules (such as "generate an `ArrayList` where a `List` is required"). It can also be addressed for specific test scenarios by analyzing the inheritance relationships during test generation for only the classes under test; then, given a method with a parameter of type `Piece`, the subtypes of `Piece` that are explicitly under test

would be generated as test data for the method while the subtypes of `Piece` that are not under test would not be.

The third issue is that constructing test data reflectively does not account for interrelationships among classes under test. For example, `Exemplar` takes instances of `Other` and `Thing` as parameters; suppose it requires that the `Other` and `Thing` passed to it be related to each other in a specific way (such as sharing an identification number or other such attribute). In that case, reflectively constructing the `Other` and `Thing` to pass to the `Exemplar` constructor will not establish that relationship. However, this is an issue that is also encountered in developer-designed test data, where complicated setup operations may be necessary; therefore, we accept it as a limitation of the reflective test data generation approach.

We will show in Section 5 that, despite these issues, the use of reflection to generate test data objects from primitive types provides a significant improvement in automatic test coverage over the original JMLUnit.

4.2 Context-Dependent Test Data

The second shortcoming we address is the lack of context-dependent test data. As previously mentioned, JMLUnit provides default sets of data for primitive types, and extension points for the developer to specify additional data values for primitive types as well as data for non-primitive types. However, it only provides one such extension point per type, per class under test. Though the extension points do allow some flexibility—they take a parameter to designate how far nested a loop is in which a type is being used, for example—they do not allow a developer to specify specific sets of data to be used in specific contexts.

The main reason to specify sets of data for specific contexts is to help contain the combinatorial explosion of tests. If two of the `String` parameters to the `Exemplar` constructor are names, and the other two must be parsed as numbers or other reference codes, using the same set of `String`s for all 4 parameters will result in many meaningless tests. Specifying a set of `String`s for the names and another set of `String`s for the numbers/reference codes allows the developer to reduce the number of meaningless tests, and thus reduce the time it takes to run the test suite.

JMLUnitNG provides extension points for the developer to specify an individual set of test data for each parameter of each method under test. These extension points have data types and method signatures embedded in their names to uniquely associate each with a context; for example, method `Exemplar.m()`, declared as `int m(int one, String two)`, would have extension points with names like `int_one_m_int_String` (`int` data to be used for the `one` parameter of the method with signature `m(int, String)`) in the generated test class. For non-primitive types, these extension points invoke the reflective data generation code described earlier by default.

In addition to these extension points, JMLUnitNG also provides "global" extension points that allow the developer to add test data for all occurrences of a given type, as in the original JMLUnit; such global extension points have

names like `char_for_all`. The test data that is actually used at runtime for a given method parameter consists of the default test data set generated by JMLUnitNG, the global test data set associated with the data type, and the test data set associated specifically with that method parameter.

The addition of custom test data sets for individual method parameters allows developers to fine-tune their test suites and to easily integrate data from external test data generators into the system.

4.3 Iterators and Lazy Test Generation

The third shortcoming we address is JMLUnit's excessive memory utilization. There are two main causes of memory utilization when running automated tests: the need to generate all the tests in a test suite before executing the suite, and the recording of information about executed tests using in-memory data structures.

Since the tests generated by JMLUnit are extremely repetitive—each method is called many times, with parameter lists generated by taking the cross product of the test data sets for its parameter types—an ideal way to execute them would be to lazily generate the parameter lists as they are needed, rather than marshaling the parameter lists for all the individual method calls in memory as part of setting up the test suite. Unfortunately, the JUnit test execution engine does not support lazy parameter list generation. While it does have the ability to run parameterized tests, where a single test method is run repeatedly with multiple parameter lists, it requires the parameter lists to be stored in a two-dimensional array in memory; this makes it impossible to save memory by parameterizing the tests.

In order to enable lazy parameter list generation, we replace the underlying JUnit engine used by JMLUnit with TestNG,[6] a Java-based test execution engine that is similar in concept to JUnit but has a different feature set. Like JUnit, TestNG supports the use of arrays as data sources for parameterized test methods; however, it also supports the use of *iterators* for this purpose. When it encounters a test method that uses an iterator as a data source, it executes the test method with parameter lists provided by the iterator until the iterator is empty. This allows us to implement lazy parameter list generation; by using iterators over primitive test data sets and the previously-discussed iterators that generate test objects of non-primitive types, we can create combined iterators that generate parameter lists for test methods while only keeping a single parameter list in memory at a time.

TestNG also supports another critical feature that helps to avoid excessive memory utilization: it allows the use of custom *test listeners* to record detailed information about executed tests, including the parameters used for testing and the exception, if any, that caused the test to fail or be skipped. Thus, instead of recording every test result in memory and processing that information at the end of a test suite's execution, as the previous version of JMLUnit does, we can record test results to disk in a streaming fashion as the tests are executed, with as much

[6] http://www.testng.org/

detail as we choose. As distributed, TestNG does record every test execution in memory—even if the default test listeners are disabled—in order to present a basic test report at the end of execution. However, with only minor changes to the TestNG source code, we were able to eliminate this in-memory recording while maintaining the ability to use other desirable TestNG features. With our modified version of TestNG, we can run test suites of essentially arbitrary size in a reasonable amount of memory, provided that there is sufficient disk space to log their results; we have successfully run hundreds of millions of tests using less than 1 GB of Java heap space.

The switch from JUnit to TestNG as a test execution environment therefore allows us to eliminate all the memory issues associated with JMLUnit. It also removes the need for a custom test runner that understands meaningless tests, because TestNG natively supports the concept of a *skipped* test; we simply record the meaningless tests as skipped, by intercepting the appropriate JML assertion errors and wrapping them in TestNG `SkipException`s. In addition, because TestNG supports functionality such as dependencies among tests and multiple forms of parallel testing, it provides a robust platform upon which to perform future automated test generation experiments.

5 Comparison of JMLUnit and JMLUnitNG

We have run our current version of JMLUnitNG on two different sets of Java classes. Both are relatively small; one is a small set of classes that implements chess pieces and the other is a set of core classes from the *Kiezen op Afstand* (KOA) Internet-based remote voting system [12] constructed for the Dutch government by the Security of Software group at Radboud University Nijmegen.

The chess piece classes are largely testable in isolation, though they have a dependency on a `Team` class[7] that is used to indicate whether each piece is black or white and to enable the pieces to determine their legal directions of movement. The piece classes, which are named for the pieces whose movements they model, have methods that take no more than 3 parameters; the majority of their methods take fewer than 2 parameters. The piece classes tested here share a common interface (`Piece`) but do not take advantage of inheritance to factor out the common functionality of chess pieces into a shared parent class; thus, they all have similar structure.

The KOA classes, by contrast, are highly interrelated, with some taking instances of multiple others as constructor and method parameters. They also have a significantly greater number of method parameters on average, making the combinatorial explosion of test method calls more pronounced. The classes in the KOA system model components of the Dutch election system: `District` represents a voting district; `KiesKring` represents a *kieskring*, which is a region containing a collection of voting districts that are counted together for the purpose of proportional representation in the lower house of the Dutch parliament;

[7] This is a `class` in the chess code tested here, because we are working with a version of JML that only handles Java 1.4 constructs; it would be an `enum` in modern Java.

Class	Total Blocks	Covered Blocks Orig	Covered Blocks New	% Covered Orig	% Covered New
`Candidate`	197	0	0	0	0
`CandidateList`	659	0	0	0	0
`District`	98	13	74	13.3	75.5
`KiesKring`	299	29	207	9.7	69.2
`KiesLijst`	431	45	173	10.4	40.1
`VoteSet`	745	0	0	0	0
Total	2429	87	454	3.6	18.7

Table 1. Results for KOA classes with JMLUnit (Orig) and JMLUnitNG (New)

Class	Total Blocks	Covered Blocks Orig	Covered Blocks New	% Covered Orig	% Covered New
`Bishop`	367	0	247	0	67.3
`King`	390	0	270	0	69.2
`Knight`	362	0	242	0	66.9
`Pawn`	403	0	273	0	67.7
`Queen`	368	0	248	0	67.4
`Rook`	360	0	240	0	66.7
`Team`	10	1	8	9.1	80
Total	2260	1	1528	0	67.6

Table 2. Results for Chess classes with JMLUnit (Orig) and JMLUnitNG (New)

`Candidate` stores information about a single candidate for office; `KiesLijst` stores a list of candidates for a particular kieskring; and `CandidateList` stores information about the entire set of candidates, across all regions, for a single election.

We use EMMA,[8] a code coverage tool for Java, to measure the coverage of the tests generated by JMLUnit and JMLUnitNG. EMMA measures coverage in terms of *basic blocks*, which are sequences of bytecode instructions without any jumps or jump targets, rather than in terms of lines of source code. When a Java program is run under EMMA, it generates a report that lists all the classes loaded by the virtual machine, their methods, the number of basic blocks in each method, and the number of those blocks that were executed during the run.

Tables 1 and 2 show the block coverage provided by JMLUnit and JMLUnitNG, based on the data in the EMMA reports. Both sets of generated tests were run with default settings and without modifying the generated code. For the chess classes, 165 tests were automatically generated by JMLUnit and 7,108 were automatically generated by JMLUnitNG; for the KOA classes, 686 tests were automatically generated by JMLUnit and 3,017 were automatically generated by JMLUnitNG. The disparity—JMLUnitNG generates fewer tests for the KOA classes than for the chess classes, while JMLUnit does the opposite—is

[8] http://emma.sourceforge.net

Class	Total Blocks	Covered Blocks	% Covered
`Candidate`	197	118	59.9
`CandidateList`	659	74	11.23
`District`	98	75	76.5
`KiesKring`	299	239	79.9
`KiesLijst`	431	266	61.7
`VoteSet`	745	167	22.4
Total	2429	939	38.7

Table 3. Results for KOA classes with JMLUnitNG and provided primitive data values

due to the fact that the constructors for the chess classes have significantly less restrictive preconditions; while the default test data generate many possible parameter lists for constructing test objects, significantly fewer of those satisfy the constructor preconditions for the KOA tests than for the chess tests.

Since JMLUnit has no way to construct objects on which to call test methods, it fails to provide any test coverage other than for object constructors that take only primitive values (or accept `null`, which JMLUnit uses as a default). By contrast, JMLUnitNG covers significant fractions of the systems under test with no developer intervention.

Adding primitive and `String` data to the JMLUnit tests, for either set of classes, does not improve their coverage because JMLUnit still does not construct test objects. Adding primitive and `String` data to the JMLUnitNG chess tests does not improve the coverage significantly, because the default values for the primitive types are sufficient to test nearly everything that can be tested by JMLUnitNG; the polymorphism limitation mentioned in Section 4.1 prevents JMLUnitNG from automatically generating useful tests for the methods that handle capturing of pieces, which take parameters of type `Piece` (an interface shared by all the pieces), or for methods like `equals`. However, adding primitive and `String` data for the JMLUnitNG KOA tests has a significant impact, as the added data can be chosen to satisfy constructor preconditions that are not satisfied by the default data. Table 3 shows that block coverage more than doubled when a few carefully-selected primitive and `String` data values were added to the test data set; JMLUnitNG generated 1,351,351 tests for that run.

The test runs with default data ran in less than 10 seconds each; however, the JMLUnitNG test run with added data required approximately 3 hours to complete the 1,351,351 tests. We believe that the execution time can be dramatically improved through optimization of the reflective test data generation process, as well as by parallelizing the test executions. However, the completion of a million-test run is itself a dramatic improvement over the original JMLUnit tool; it would have exhausted the available 16 GB of Java heap space during the attempt and generated no results, while JMLUnitNG used less than 768 MB of heap space and reported that all the tests passed.

6 Related Work

As previously mentioned, considerable research has been (and continues to be) devoted to automatic test generation, most of it to the generation of test data rather than test oracles. We have insufficient space here to give even a complete overview of the current state of the art. We thus describe only the most closely related of the existing automated test generation systems.

Test oracles can be derived from a behavioral specification of the SUT, such as structured documentation [15], a formal model [9], or inline specification statements written in languages such as JML (as we have used here). Regardless of the type of behavioral specification, the basic idea is the same as we have employed: a test oracle is generated for each unit based on the specification of that unit; tests that are run with data that would violate the unit's requirements (preconditions, assumptions) are ignored, and a test is considered to pass if the unit's specification is not violated by the test execution.

Most automated test data generation falls into one or more of the following categories: *randomness-based*, where test data are generated randomly; *optimization-based*, where test data are optimized over multiple test runs based on coverage observations; *code-driven symbolic execution-based*, where symbolic execution [11] is used to compute test data that will exercise particular execution paths of the SUT; *specification- or model-based*, where constraint solving is used to generate test data based on a logical analysis of a specification or model of the SUT; and *verification-based*, where test cases are generated from attempts to formally verify the SUT. The latter two are most closely related to our approach.

Specification- and model-based test data generation methods, implemented in tools such as BZ-TT [1], JML-Testing-Tools [3] and UniTesK [4], use a logical analysis to compute partitions of the variables that fulfill the explicit case distinctions present in a formal specification or model of the SUT. Once the partitions have been computed, constraint solving or model finding is used to find concrete test data in each partition.

Verification-based test data generation (hereafter, *VBT*) is a recent development, based on the idea of generating test cases from attempts to verify systems with formal specifications [10]. VBT uses symbolic execution, with termination being enforced by a bound on the number of times loops and recursions are unwound; it differs from code-driven symbolic execution-based methods by generating test data from path condition formulae encountered at termination nodes in the symbolic execution tree. The VBT approach works well for code with simple branching statements (if...then, switch/case, constant-bounded loops) but not as well for code with generalized loops or recursion, because only a limited number of loop iterations and only a limited recursion depth can be dealt with. VBT has been implemented in the KeY verification system [2] and in Kiasan/KUnit [8]. A uniform framework for verification and testing has been formalized in HOL/Isabelle for a small target language [5].

JMLUnitNG is complementary to, not competitive with, the test generation methods and tools described above. While these methods and tools are relatively heavyweight, using automated theorem provers, constraint solvers and symbolic

execution engines, JMLUnitNG is extremely lightweight, using only the TestNG framework and Java's reflection mechanism. It is an instant replacement (and improvement) for developers who already use JMLUnit, and a one-step addition to the software build process for developers who use JML but have not yet adopted JMLUnit. It is easy to use, and the principles underlying its operation are easy for typical software developers and students to understand regardless of their level of experience with JML specifications and tools. For more advanced developers, it can also be used in conjunction with more heavyweight methods; rather than manually creating context-dependent test data sets for the JMLUnitNG test oracles, or relying solely on the default data sets and reflective data generation, developers can create their data sets using one or more other test data generation tools.

7 Conclusion

We have presented JMLUnitNG, a new unit test generation and execution framework inspired by the original JMLUnit tool and based on a modified version of the TestNG unit testing framework for Java. The current implementation has some shortcomings; as a proof of concept, it was directly evolved from the original JMLUnit and is based on the Common JML tool suite, so it cannot be used on code that contains modern Java constructs such as generic types. It does not contain solutions for two of the issues—cyclic dependencies and polymorphism—discussed in Section 4.1. When generating test data, it cannot reflectively construct instances of classes that have no public constructors, such as those that rely on factory methods. We have already designed and partially implemented a new version of the tool, independent of the Common JML tool suite, to address all these issues.

Despite these shortcomings, we consider our initial experiments with JMLUnitNG to be quite successful; the ability to generate and rapidly execute millions of tests and the automatic generation of test data of non-primitive types are substantial improvements over the functionality provided by the original JMLUnit, and the resulting benefits can be easily realized in any project that currently uses JMLUnit for specification-based testing. Moreover, JMLUnitNG provides significant new developer flexibility, including the ability to specify context-dependent test data. As such, it is not only an improvement over the original JMLUnit, but also a sound foundation for future test data generation experiments.

Acknowledgements

A portion of this work was funded by a 2008–09 award from the University of Washington Tacoma Chancellor's Fund for Research & Scholarship. In addition, the authors would like to thank Dr. Joseph R. Kiniry for his role in initial discussions about JMLUnitNG and his useful comments during both its development and the writing of this paper.

References

1. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N.: BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In: Formal Approaches to Testing of Software (FATES) 2002, Workshop of CONCUR'02. Brno, Czech Republic (Aug 2002)
2. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software: The KeY Approach. No. 4334 in Lecture Notes in Computer Science, Springer-Verlag (2007)
3. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: JML-Testing-Tools: A symbolic animator for JML specifications using CLP. In: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Edinburgh, U.K. (Apr 2005)
4. Bourdonov, I.B., Kossatchev, A., Kuliamin, V.V., Petrenko, A.: UniTesK test suite architecture. In: International Symposium of Formal Methods Europe (FME). Copenhagen, Denmark (Jul 2002)
5. Brucker, A.D., Wolff, B.: Interactive testing with HOL-TestGen. In: Fifth International Workshop on Formal Approaches to Testing of Software (FATES). Edinburgh, U.K. (Jul 2005)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (Feb 2005)
7. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2002. Lecture Notes in Computer Science, vol. 2374, pp. 231–255. Springer-Verlag (2002)
8. Deng, X., Robby, Hatcliff, J.: Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In: Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART). pp. 3–12. Windsor, UK (September 2007)
9. El-Far, I.K., Whittaker, J.A.: Model-based software testing. Encyclopedia on Software Engineering (2001)
10. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Tests and Proofs, First International Conference (TAP). Zürich, Switzerland (Feb 2007)
11. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (July 1976)
12. Kiniry, J., Morkan, A., Cochran, D., Fairmichael, F., Chalin, P., Oostdijk, M., Hubbers, E.: The KOA remote voting system: A summary of work to date. In: 2nd International Symposium on Trustworthy Global Computing (TGC). Lucca, Italy (2006)
13. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. In: Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO) 2002. Lecture Notes in Computer Science, vol. 2852, pp. 262–284. Springer-Verlag (2003)
14. Meyer, B.: Object-Oriented Software Construction, 2nd Edition. Prentice-Hall (1988)
15. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering 24(3), 161–173 (March 1998)
16. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4), 366–427 (Dec 1997)