ABSTRACT

There are several problem areas that must be addressed when applying randomization to unit testing. As yet no general, fully automated solution that works for all units has been proposed. We therefore have developed RUTE-J, a Java package intended to help programmers do randomized unit testing in Java. In this paper, we describe RUTE-J and illustrate how it supports the development of per-unit solutions for the problems of randomized unit testing. We report on an experiment in which we applied RUTE-J to the standard Java TreeMap class, measuring the efficiency and effectiveness of the technique. We also illustrate the use of randomized testing in experimentation, by adapting RUTE-J so that it generates randomized minimal covering test suites, and measuring the effectiveness of the test suites generated.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Testing Tools

Keywords

Randomized Testing, Unit Testing

1. INTRODUCTION

Unit testing is the practice of testing small portions of programs, such as methods, groups of methods, or classes. Unit testing is becoming more widespread as a result of the popularity of tools like JUnit [5]; for instance, one recent survey notes that 79% of Microsoft developers write unit tests [34].

A unit test typically consists of a sequence of calls to the methods under test, each call possibly preceded by some code setting up the parameters to the method, and each call possibly followed by some code checking whether the method has performed correctly. We use the phrase randomized unit testing to refer to any practice in which there is some random element to the selection of parameters and/or the selection of methods (which methods to call, how many method calls to make, and calling order).

There are several main problems that we have to solve in order to perform randomized unit testing for a given unit. These include identifying optimal ranges for scalar method arguments, constructing complex method arguments, and specifying correct behaviour. Many of these problems are shared with bounded exhaustive unit testing, and some are shared with other forms of unit testing. Many solutions have been proposed and implemented for these challenges, but as yet the “pushbutton” or fully-automated solutions only address part of the problem or work for only some units.

We believe that for the time being, in many cases it is more realistic for programmers to solve these problems on a unit-by-unit basis than to expect a fully automated solution that works for all units. We have therefore implemented RUTE-J, a Randomized Unit Testing Engine for Java, which supports the development of randomized testing for Java units. RUTE-J facilitates such things as user selection of number of runs and run length, performs Zeller-Hildebrandt minimization of failing test cases, and handles test case storage and retrieval. It also provides an API that helps users define and refine their own unit-specific solutions to the problems of randomized unit testing. We believe that such a tool will help programmers to explore and reap the benefits of randomized unit testing more than they have been able to in the past.

Randomized unit testing has two main applications. One is to the actual testing of units; the other is to experimental software engineering, specifically to the generation of “arbitrary” test suites that have given properties. We explore both applications in this paper.

In Section 2, we elaborate on the problems of randomized unit testing and the past solutions. In Section 3, we describe RUTE-J and illustrate its use, using methods of the standard Java TreeMap class as an example. In Section 4, we report on the results of an experiment to measure the test effectiveness of the TreeMap example, an indication of how randomized unit testing would fare in the real world. In Section 5, we apply RUTE-J to an experimental end, adapting it so that it generates random minimal test suites that achieve statement coverage on the TreeMap unit; we show that these minimal test suites display rather poor effectiveness, reinforcing past research that suggests that achievement of statement coverage alone is not a guarantee of effectiveness. In Section 6, we conclude and discuss future directions.

2. PROBLEMS AND PROPOSED SOLUTIONS

There are five main problem areas that must be addressed when applying randomized testing to unit testing. These are (1) selection of optimal data ranges for scalar method arguments such as integers, (2) construction of string arguments, (3) construction of
complex (non-scalar) arguments for method calls, (4) evaluation of test results for correctness, and (5) selection of optimal run length, number of runs and method probabilities. In this section, we will elaborate on each of these problem areas and describe the solutions (if any) attempted in the past, either in randomized testing or in other related approaches to testing or analysis.

2.1 Selection of Optimal Scalar Ranges

When doing randomized testing, we must select ranges from which to draw scalar (e.g. integer) arguments for methods. These ranges can have a profound effect on the effectiveness and efficiency of the testing.

For instance, say we are testing the “add key” and “delete key” methods of a B-tree implementation of a dictionary unit, where the keys are 32-bit integers and each key can have a unique piece of data associated with it. If we always select the integer key from the range 0-3, then every tree that we construct can have a maximum of 4 nodes in it, and many lines of the B-tree code (for instance, code for splitting the root node of a 2-level tree) will remain uncovered and thus untested. However, if we select the integer key from the full range available (0-2^32 – 1), we will typically have to execute a long sequence of successful adds and unsuccessful deletes before our first unsuccessful add or successful delete, thus raising an unacceptable tradeoff between coverage and efficiency of the testing. We have found for one B-tree implementation that a range of 0-31 is efficient and covers all the code. Obviously, this “magic number” appears nowhere in the code and thus cannot be found by a simple analysis of the constants in the program.

JCrasher [10] generates integer method parameters by selecting from the three values -1, 0 and 1. Some analysis-based approaches for generating unit test cases rely on techniques such as symbolic evaluation in order to identify integer inputs that cause code to be covered. For instance, if x is a parameter and the first if condition in the method under test is if (x>3), then an integer on either side of 3 is chosen. However, conditions can in general be very difficult to solve by analytic means, or even impossible (in cases such as Diophantine equations).

When testing units with a model checker such as Java PathFinder [17] or CUTE [31], the standard technique is for the user to specify ranges for integers via calls to a utility routine, such as PathFinder’s Verify.random(). If formal analysis of the model does not succeed in identifying useful integer parameters, then every element of the range is tested exhaustively or randomly. The user uses their own reasoning, intuition or trial and error to determine the ranges, making more intelligent choices as they develop experience with the tool and technique.

2.2 Construction of String Arguments

A similar but even more difficult problem exists here, since often the best choice of a string argument is not random at all, but rather a repetition of a string argument to a previous method call, or an argument that is a sentence of some grammar (or, for testing error checking, an argument that is not quite a sentence of the grammar). We know of no approaches to this problem except those that assume that we will specify an explicit grammar for the argument.

2.3 Construction of Complex Arguments

Some arguments, even though not integer or string themselves, can be constructed by randomly choosing a finite number of integers and/or strings. For instance, we may be able to construct an “employee” object that is random enough for our purposes by choosing a random “name” string and a random “salary” integer. However, other arguments cannot be constructed from a finite number of random integer and string choices. We refer to these arguments as “complex arguments”.

Examples of complex arguments include data structures of arbitrary size; for instance, we may have a vector of values of arbitrary length to enter into a container. They also include values that have been returned from methods under test and must be fed back to other methods under test; for instance, in the C++ STL Vector class, we can extract two iterators from a vector, move them to given points within the vector, and then call a method that removes everything between the two iterators.

When using Korat [7] and TestEra [20], some complex arguments can be used by specifying a method that defines the correct form of the arguments; the tool then constructs values of the given form. In the work of Leow et al. [25], the user defines criteria that complex arguments must meet, and formally describes the effects of method calls. Then, a planning algorithm is run that attempts to construct the arguments by sequences of method calls.

When doing model checking of units using tools such as Java PathFinder, the user must write a main program that calls the methods under test. Thus, implicitly the user is the one that decides how complex arguments are to be constructed, by writing the code to do so.

2.4 Evaluation of Test Results

When doing any kind of automated unit testing, we must tell the system how to evaluate whether a test case has done the right thing. When we construct single unit test cases with frameworks like JUnit [5], this is a relatively simple matter of asserting boolean expressions that we think should be true after the test case has been run. However, for any approach in which test cases are constructed by a tool, some more general information is needed.

The rich research area of specification-based testing addresses this issue. This research includes that by Doong and Frankl [12] and Antoy and Hamlet [4], in which users give algebraic specifications of units, and that of Hoffman and Strooper [19], in which users specify transitions between selected points in the state space of classes under test. JML [8] is a language in which users can specify the behaviour of methods, embedded in comments in the Java code for the methods.

2.5 Selection of Test Case Construction Parameters

Since unit test cases consist of sequences of target method calls interspersed by parameter-setup and result-checking code, it is natural to measure the length of a unit test case as the number of target method calls in it. One of the intentions of randomized unit testing is that we are able to generate test cases of any length. But how long is long enough? Returning to the B-tree dictionary example, regardless of the range of keys, a sequence of 4 or fewer method calls cannot add more than 4 keys to the tree, and thus may leave code uncovered for the same reason. A sequence of 10,000 method calls may or may not cover all the code, but may take an unacceptably long time to run.

Related to this question is the question of how many randomized test cases to generate. For instance, a container data structure may allow or disallow duplicate keys, based on a parameter in the constructor. If we build a random test case as a single constructor call followed by a sequence of method calls, then one test case may be inherently unable to achieve our black- or white-box coverage goals. The easiest solution to this problem is to run more than one randomized test case; but how many runs is enough, and how does this question interact with test case length?

A final related question concerns the weights assigned to target
method calls in generation of the test cases. By the “weight” of method \( m \), here we mean the probability that any random selection of the next method to call will select \( m \). Returning to the container class example, if the container has one form of method call for adding a key but several for deleting a key (for instance, delete key, delete keys in this range, delete all keys), and all methods are weighted equally, then a randomly chosen method will be much more likely to delete keys than to add them. This may exert downward pressure on the size of the container, again raising problems with coverage. Antoy and Hamlet report this problem when doing randomized testing of a vector class [4].

In the randomized unit testing framework Jartege [29], these parameters are specified explicitly. In model checking approaches, the run length is implicitly or explicitly specified by the depth to which the model is checked.

3. RUTE-J

In this section, we describe our package RUTE-J, which helps programmers find per-unit solutions to the problems of randomized unit testing. We use a running example to illustrate the concepts. We first discuss the central concept of test fragment collection, the piece of Java code that a programmer must write for a unit in order to use RUTE-J. We then describe how the user runs RUTE-J on the test fragment collection. Finally, we revisit the five main problem areas discussed in the last section and summarize how RUTE-J helps programmers find their own solutions to them for the units they want to test.

As our example unit, we take nine methods from the Java TreeMap class, a red-black tree implementation that is a standard part of the Java library and is often used for studies in formal verification (see, for instance, [7, 35]). These nine methods (size, containsKey, containsValue, get, firstKey, lastKey, put, remove, and clear) are the main insertion, deletion and retrieval operations.

3.1 Test Fragment Collections

The main task that a programmer performs in testing a unit with RUTE-J is writing a “test fragment collection”. A test fragment collection is a subclass of TestFragmentCollection, an abstract class with helper methods. Every test case created corresponds to a new instance of the test fragment collection.

3.1.1 Fields and Initialization

Figure 1 shows the first part of a test fragment collection we wrote for the TreeMap methods. Line 2 contains a declaration and initialization for a TreeMap object, which for a generated test case will be the receiver of all the method calls. Lines 3-7 contain declarations for fields that will contain predictions for the three pieces of information that we can extract from the TreeMap: the size (number of keys) in the TreeMap, whether the TreeMap contains a given key, and what the “value” associated with the key is.

Although the TreeMap methods can take any Object as either the key or the value, we chose in this test fragment collection to use only Integers in the range 0-31 as both keys and values. We did this because it facilitates the test result checking: we want predictedContainsKey[i] to be true and predictedValue[i] to be \( j \) if \( i \) is a key in TreeMap associated with value \( j \). The range 0-31 is arbitrary, but is based on our past experience with other data structure units; if we find we need a wider or narrower range in order to cover more code, then we can get it by changing the constants KUB and/or VUB.

The initialize() method is called by RUTE-J at the start of every newly-generated test case (thus it is the analogue of the setUp() method in JUnit test cases). In this collection, the declarations on lines 2-7 and the initialize() method on line 9-14 state that our initial prediction is that TreeMap has a size of 0 and contains no keys. The setWeight() call on line 13 will be discussed below.

3.1.2 Test Fragments

The rest of the TreeMap test fragment collection consists of declarations of the test fragments themselves. A RUTE-J test fragment is a method which returns void, whose name starts with the characters tf_, which takes only parameters of class SelectedValue and ResultValue, and which throws a Throwable if it has detected a failure of the unit under test. SelectedValue parameters represent values which will be randomly selected by the test fragment, and ResultValue parameters represent the results of method calls.

Figure 2 shows a sample test fragment. In our TreeMap test fragment collection, we chose to write one test fragment for each method under test, and to name the test fragment after the method under test (thus tf_put for the method put, line 15). The put() method of TreeMap takes a key and a value (both of class Object) as parameters, and ensures that the key is now associated with the value in the tree. If there was a previous value associated with that key, then it returns that value; otherwise put() returns null. The central call to the TreeMap.put() method is on line 21. It is preceded by selection of the two integer parameters (lines 16-19) and followed by some assertions about the return value (lines 23-27) and an updating of the predicted values (lines 28-32).

Note that we pass the randomly selected target method parameters key and value back to RUTE-J by explicitly declaring them as the SelectedValue parameters keyV and valueV of the test fragment. This allows RUTE-J to store the values in a deterministic “test case” object that it can write to disk and restore for re-running.

Note also that we use Java 1.4’s assert facility to assert properties of the return values on lines 25 and 27. In the style of test fragment collection construction that we use, we examine only the values returned, parameters modified and exceptions thrown by the method under test in the test fragment for that method. In tf_put we do not, for instance, make a call to TreeMap.get() in order to check whether the key was added. Instead, we simply update our predicted information, and trust that the randomized test case construction procedure will frequently call the test fragment tf_get immediately after the test fragment tf_put, yielding the same effect. This style ensures that truly any sequence of calls can be executed by the engine, including possibly failing sequences that might be missed if attention is restricted to sequences in which every put is followed by a get.

3.1.3 Fragment Weights

We can now return to line 13 of Figure 1. We have in our test fragment collection one test fragment for each method under test, but there is one method (put) that increases the size of the TreeMap and two (remove and clear) that decrease it. Moreover, clear removes all keys. Leaving all test fragments of equal weight will cause the size of our TreeMap to usually be very small, compromising coverage, since we will have an average of only one put and one (attempted) remove before all keys are removed with clear.

In initialize(), we therefore set the weight of tf_clear to 5. All test fragments have a default weight of 100, but the weights can be adjusted with set_weight. Setting the weight
public class RuteTreeMapTest extends TestFragmentCollection {
    private TreeMap treeMap = new TreeMap();
    private int predictedSize = 0;
    private int KUB = 31; // Upper bound for keys
    private int VUB = 31; // Upper bound for values
    private boolean[] predictedContainsKey = new boolean[KUB+1];
    private int[] predictedValue = new int[KUB+1];
    
    public void initialize() {
        for (int i=0; i<=KUB; i++) {
            predictedContainsKey[i] = false;
        }
        setWeight("tf_clear", 5);
    }

    public void tf_put(SelectedValue keyV, SelectedValue valueV) {
        keyV.selectRandomInt(0, KUB);
        int key = keyV.intValue();
        valueV.selectRandomInt(0, VUB);
        int value = valueV.intValue();

        Object prevValueO = treeMap.put(new Integer(key), new Integer(value));
        
        if (predictedContainsKey[key]) {
            Integer prevValueI = (Integer) prevValueO;
            assert prevValueI.equals(new Integer(predictedValue[key]));
        } else {
            assert prevValueO == null;
            predictedSize++;
        }

        predictedContainsKey[key] = true;
        predictedValue[key] = value;
    }
}

Figure 1: Listing of the first part of the test fragment collection developed for the Java TreeMap unit.

Figure 2: An example test fragment from the test fragment collection for TreeMap.
of \texttt{tf\_clear} to 5 therefore means that when selecting an arbitrary next test fragment to call, we will be 20 times more likely to call \texttt{tf\_add} or \texttt{tf\_remove} than \texttt{tf\_clear}. In doing so, we are making a judgement call that the decreased ability to find bugs in \texttt{clear} is less likely to decrease the effectiveness of the testing overall than an inability to achieve coverage. Antoy and Hamlet made a similar decision to resolve their vector class testing problem [4].

### 3.2 Running RUTE-J

To use RUTE-J, we start its Java Swing-based GUI, giving the \texttt{TestFragmentCollection} subclass as an argument. The menu item “Run new test cases” allows us to ask RUTE-J to construct random new test cases. We can ask for a single test case of a given length, or a given number of test cases of a given length; we can also ask it to construct new random test cases starting at a given length and going up in length by a given amount until a stopping condition is reached (see Figure 3). This last option is useful when we are not sure of the minimum test case length that is likely to result in a failure.

To construct a test case, RUTE-J creates a new instance of the test fragment collection, calls its \texttt{initialize()} method, and then randomly selects and calls a number of test fragments equal to the requested length. Each “test case” thus constructed can be described as a sequence of calls to test fragments, each with its own associated randomly-selected method parameter values, passed back in the \texttt{SelectedValue} test fragment arguments.

In contrast to JUnit, randomly generated successful test cases are considered unimportant and are discarded. As soon as RUTE-J finds a failing test case, however, it stops the process of generating test cases and reports a failure. The user is then able to view the test case and store it on disk. We can later use RUTE-J to retrieve the test case from disk and re-run it.

At the point of finding a failing test case, the user can view it, or minimize it at the click of a button using Zeller and Hildebrandt’s algorithm [36]. We have shown [23] that this minimization is an effective, efficient way of focusing randomized unit tests down to the essential sequence of method calls that caused the failure.

### 3.3 Case Studies

We have applied RUTE-J to a number of freely-available Java classes of various sizes, some data structures and some not. These included the JUnit \texttt{IMoney} interface, with its two implementations \texttt{Money} and \texttt{MoneyBag} [5]; the \texttt{Node} class from JTidy, an HTML syntax checker and prettyprinter that has been a subject program in previous studies, e.g. [24]; and the tokenizer classes from JTopas, a tokenizer and parser package that has also been used in previous studies, e.g. [11]. Our major study was of 4 interfaces and 9 classes containing a total of 410 methods, from a large commercial Java data structure package.

Despite these packages being well-studied and stable, we found a total of 5 faults with no workaround and 22 faults with possible workarounds among the packages tested. For instance, using RUTE-J we found a fault essentially arising from a design problem in the JUnit \texttt{MoneyBag} class, despite the fact that the fault had existed through almost three years and over 600,000 downloads of the JUnit package from SourceForge. Kent Beck, one of the Sourceforge project admins of JUnit, has since confirmed that the fault is a fault [6].

Our success in finding previously undiscovered faults in these units supports the claim that randomized testing using RUTE-J is an effective testing technique in practical settings. The results of the case studies are reported in more detail in [2].

### 3.4 Support for Problem Resolution

We are now in a position to summarize how RUTE-J helps the programmer solve the problems with randomized testing discussed in section 2.

1. Selection of optimal data ranges for scalar method arguments such as integers: As when using Java PathFinder or CUTE, the user selects ranges and calls a utility method to make the random choice. As with PathFinder and CUTE, the user must use reasoning, intuition and trial and error in order to come up with the optimal ranges; however, the fast turnaround time of randomized unit testing (see Section 4) makes the trial and error cycle more feasible. The user can set ranges, compile, run, generate a coverage report and see if they have achieved higher coverage in a time on the order of tens of seconds, in the units that we studied. We performed similar tasks with Java PathFinder and found that the cycle time was on the order of tens of minutes to hours [2].

2. Construction of string arguments: RUTE-J also has a utility method that returns a random string, allowing the user to specify an upper and lower bound on the length of a string, and a set of characters from which the characters in the string can be drawn. If something other than a totally random string is required, then the user can write test fragments that occasionally randomly perturb a \texttt{String} field of the \texttt{TestFragmentCollection} subclass, and then occasionally give the randomly-perturbed string to the method under test. RUTE-J is also compatible with approaches in which strings are generated as sentences of a grammar, although it gives no explicit support for this. For instance, the user can express the grammar as a state machine, and write a test fragment that uses one of the well-known algorithms for generating covering inputs for a state machine [22], again storing the result in a \texttt{String} field.

3. Construction of complex (non-scalar) arguments for method calls: Although the \texttt{TreeMap} example does not involve any complex arguments, other units tested with RUTE-J [2] have.

The standard technique, as with strings, is to write additional test fragments that do not directly call the methods under test, but rather randomly perturb complex arguments stored as fields, which are used as the actual arguments to the methods under test in other test fragments.
A special case of this involves methods in a given class that expect arguments or return values of the same class, such as copy constructors and \texttt{clone()} methods. For instance, say that we wanted to handle the \texttt{TreeMap.clone()} method in the \texttt{TreeMap} test fragment collection. To do this we could use an array of \texttt{TreeMaps} rather than a single \texttt{TreeMap}, and add an extra argument to all test fragments, representing the index of the \texttt{TreeMap} which will be used as the receiver of the method call. We would then add a test fragment which clones one of the \texttt{TreeMaps} in the array and stores the result in another \texttt{TreeMap} in the array. This approach would ensure that we test the fundamental required property of a clone (namely that changes to the clone do not change the original), by subjecting the original and the clone to random sequences of changes.

(4) Evaluation of test results for correctness: As with JUnit, Java PathFinder, CUTE, and other approaches to unit verification, we specify correct behaviour using normal code, usually with recourse to an assertion method or operator. It should be possible to use other inline specification languages, such as JML, or offline analysis of test case results, but we believe that most programmers will find Java 1.4 assertions sufficient.

(5) Selection of optimal run length, number of runs and method weights: The RUTE-J GUI allows users to select run length and number of runs easily, and even to progressively increase the run length until time or length limits are reached. The method weights are adjusted with calls to \texttt{setWeight()} in the \texttt{initialize()} methods of test fragments.

To summarize the summary: while fully-automated solutions to parts of these problems are possible for many units under test, in this research we avoid restricting the units under test considered, by supporting the basic framework of test fragment collections and facilitating problem resolution by programmers and simple GUI controls. We believe that this solution achieves a useful level of automation without sacrificing generality.

4. EFFECTIVENESS AND EFFICIENCY

In order to check whether earlier results of randomized unit testing in C would carry over to Java when using RUTE-J, we replicated aspects of experiments we had carried out in the past for C code [3, 23]. Our subject software for these experiments was the \texttt{TreeMap.java} class discussed above; the source code relevant to the methods studied was about 760 LOC.

We used the test fragment collection for the main nine data structure manipulation methods, and generated mutants of \texttt{TreeMap.java} to represent faulty versions of software. (Recent work [1] suggests that generated mutants can exhibit similar behaviour to real faults.) We then ran RUTE-J on the mutants, measuring its effectiveness by how many non-equivalent mutants it detected and its efficiency by CPU time. In this section, we detail the procedure that we followed in the experiment, discuss the results with respect to the effectiveness of the testing and the efficiency of the technique, and then conclude and discuss threats to validity.

4.1 Procedure

We developed the straightforward 154-LOC test fragment collection discussed above in about 2 hours. During this time, four bugs in the test fragments were identified and corrected.

For the last version of the test fragment collection, we performed various runs that requested RUTE-J to generate a total of 454 test cases having an average length of 273 test fragment calls. None of the test cases now reported a failure. We measured the coverage with the open-source JCoverage-based tool Cobertura; the test cases had covered 45% of the lines and 50% of the branches in the entire source file. We inspected the coverage report visually and confirmed that all uncovered code was unreachable from any sequence of calls to the nine methods, concluding that our random data ranges were the right size for the methods we wanted to cover.

816 mutants were generated from the \texttt{TreeMap} source code file. The mutation operators were the four that were used in [1]; that is, "replace an integer constant", "replace an operator by another operator", "negate a decision", and "delete a statement". Each mutant was different from the original by text on only one line. 163 of the generated mutants could not be compiled, and an additional 376 were mutants in which the changed line was from code that we did not attempt to cover, leaving 277 mutants that could potentially contain faults that we had a chance of detecting. (See the left-hand pie chart in Figure 4, where "nocompile" indicates non-compiling mutants and "uncovered" indicates mutants in uncovered code.)

Using the RUTE-J text interface, we ran the test fragment collection on each of these 277 mutant versions, for each mutant requesting the GUI’s default progressive action (generate new test cases starting at length 50 and going up by length 10 until 500 fragment calls were reached).

4.2 Effectiveness

Of the 277 mutants, RUTE-J forced failures and generated failing test cases for 182 of them. An additional 14 mutants timed out after 30 seconds of real time; these were presumably mutants that expressed their failure as an infinite loop, but while the GUI allows the user to break infinite loops and save the failing test case, the text interface does not yet have this facility. We conclude that RUTE-J forced failures in 196 of the mutants.

We inspected the remaining 81 mutants. Of these, we classified 10 as "absolutely equivalent", meaning that the mutant program behaved exactly as the original (for instance, when "\texttt{cmp < 0}" was changed to "\texttt{cmp <= 0}" in a place where \texttt{cmp} could not be 0). We classified a further 6 as "effectively equivalent", meaning that they changed code that affected only code that we did not attempt to cover. (See the right-hand pie chart in Figure 4.)

The remaining 65 mutants were in two methods (\texttt{fixAfterInsertion} and \texttt{fixAfterDeletion}); they affected the colours of nodes and the decision about whether or not to rotate nodes in order to rebalance the tree. Other mutants of these methods were detected, as were all the mutants of the node-rotation methods themselves. Thorough visual inspection of these 65 mutants did not reveal any reason to believe that they were non-equivalent. Some may have in fact been semantically equivalent. However, some may have been equivalent only in the weaker sense that they could produce trees that were not valid red-black trees, but were still valid binary search trees; and that we could not detect these mutants because there was
no method in TreeMap for checking whether the tree is of a valid form, and therefore no test fragment could be written to check this either. However, to be safe we make no judgement about these mutants\textsuperscript{1}.

If the 196 mutants for which RUTE-J forced failures are all the non-equivalent mutants of TreeMap that we generated, then by definition our test fragment collection forced failures for 100\% of the non-equivalent mutants. If all of the 65 rebalancing mutants were in fact non-equivalent, then the number falls to 75.1\%. The true number is somewhere between those extremes.

4.3 Efficiency

We also collected data on system CPU time for all runs. All timing information is from a Sun Ultra 2/2296 with 384Mb of main memory and two 296 MHz UltraSPARC-II processors, running SunOs 5.8.

The average CPU time taken for mutants in which no failure could be detected (i.e., those that ran test cases of length 50, 60, 70, \ldots, 500 without forcing a failure) was 1.97 seconds, and none of those mutants took more than 2.1 seconds. The average CPU time taken for non-timeout mutants in which a failure was detected was 1.04 seconds, with none taking more than 1.58 seconds. The average CPU time taken for mutants that timed out was 19.94 seconds, but this reflects the fact that these were infinite-loop mutants and that the infinite loop ate up this amount of CPU time.

Figure 5 shows, on the left, a box-and-whisker plot of the time taken for all the non-timeout failing mutants. The whiskers represent the maximum and minimum, and the box represents the first and third quartiles.

We also re-ran the experiment, requesting RUTE-J to do Zeller/Hildebrandt minimization of any failing test cases that it found. The right-hand box in Figure 5 shows the amount of CPU time taken for this for the non-timeout failing mutants; the average time was 1.26 seconds, only 21\% longer than when minimization was not done.

Figure 6 shows the effect of the minimization in box-and-whisker plots, with the left-hand plot showing the lengths of the original failing test cases and the right-hand plot showing the lengths of the minimized test cases. The average length of original failing test case for the non-timeout failing mutants was 32.52 method calls, with the maximum length of failing test case being 308. When minimization was done, the average length of failing test case was 3.14 method calls, with no test case longer than 14 method calls.

Our (subjective) experience is that minimized failing test cases are much more useful in the debugging process than unminimized ones. For instance, the tracing of the fault in the JUnit MoneyBag class (Section 3.3) was made much easier by being able to generate many small failing test cases that pinpointed the problem to the interaction of one of the methods subtract and negate with the method appendTo. The quantitative results of this section indicate that we can achieve these benefits to debugging with only a small investment of extra time.

4.4 Conclusions and Threats to Validity

We conclude that as well as being an effective method of forcing failures, RUTE-J with the test fragment collection that we developed was efficient in finding and minimizing failing test cases for the TreeMap mutants.

Threats to validity include the difficulty of generalizing from behaviour on mutants of one Java class to faults in general Java units. We used the mutant generator from [1], which was designed for C programs; although that paper concluded that the mutants behaved very similarly to real faults, we cannot necessarily conclude that our mutants behaved similarly to real faults for Java units. In addition, we cannot be certain that our classification of mutants into equivalent and non-equivalent was always correct, especially for the rebalancing mutants.

5. EFFECTIVENESS OF MINIMAL TEST SUITES

We now illustrate the use of randomized testing for research purposes by discussing an experiment that we performed on the effectiveness of covering test suites. Reusing the RUTE-J code and Cobertura, we wrote an application that generates random test suites that achieve line coverage goals without making extra calls not needed for achieving those goals. These random test suites can be used to represent arbitrary covering test suites that may be gener-
ated by a tool or by a human tester guided only by test case length and line coverage. We found that the generation of covering test suites was efficient and was able to achieve good coverage. However, consistent with some results in the literature, the covering test suites were not very effective at finding faults, reinforcing the message that both black-box and white-box techniques are important to use in tandem.

5.1 Procedure

We wrote a new main program, called Autocov, which reused much of the RUTE-J code (about 1KLOC extra code, compared to 5.6KLOC for RUTE-J). In RUTE-J, the running of a test case returns a value of false if the test case fails, whether it is a new test case or one being re-run. In Autocov, a test case returns a value of false also if it covers lines of code not yet added to a set $C$ of “already-covered” lines of code. Thus, when generating new test cases, Autocov stops when new lines of code have been covered, and when minimizing them. Autocov maintains the property that the test case covers new lines of code (though the minimized test case may cover fewer new lines of code than the original).

When given an initial test case length $n$, step size $s$ and maximum length $m$, Autocov executes the following algorithm:

1. Let the set $C$ of covered lines be the empty set.
2. Repeat:
   a. Generate test cases starting at length $n$ and going up in length by $s$ until a test case covers lines not in $C$, or until a test case of length $m$ has been generated.
   b. If the last test case generated does not cover new lines of code, then exit.
   c. Minimize the last test case.
   d. Output the minimized test case.
   e. Add to $C$ the lines of code covered by the minimized test case.

Line coverage is measured, as in the last section, with Cobertura. We minimized the new test case in step (c) for two reasons. First, we wanted to simulate the behaviour of tools that generate covering test suites, which typically pick an element to cover, generate the smallest test case that would cover that element, and then move on to the next element. Second, without minimization, the tool generated long test cases which made many method calls not needed for the coverage goals; these test cases did not realistically represent test cases that would have been found either by a human tester or by a non-randomized tool.

Autocov therefore outputs a sequence of test cases, each of which covers lines of code not covered by the previous test cases. If we set $n$, $s$ and $m$ to values that we judge (based on randomized testing) to be likely to achieve maximal coverage, then we should obtain a set of test cases that achieves good coverage in a small number of test cases with small length.

We refer to these test suites briefly as “minimal covering test suites”, although they are minimal only in the sense that each test case has been reduced to cover the minimal amount of extra code. Minimal covering test suites are interesting experimentally because (a) we can generate many different ones, and (b) each one represents a possible test suite that might be generated by a human or automatic test suite generator guided solely by test case size and line coverage. These test suites can be used as a technique of eliminating experimenter bias, in experiments such as those pioneered by Frankl and Weiss [14] to compare coverage criteria.

### Table 1: Statistics, with 95% confidence intervals, of the experiment generating minimal covering test suites.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Mean</th>
<th>Conf. Int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time to generate</td>
<td>18.98</td>
<td>(18.45, 19.51)</td>
</tr>
<tr>
<td>Number of test cases</td>
<td>46.12</td>
<td>(45.72, 46.52)</td>
</tr>
<tr>
<td>Total number of method calls</td>
<td>158.16</td>
<td>(155.75, 160.57)</td>
</tr>
<tr>
<td>Number of lines covered</td>
<td>247.92</td>
<td>(247.76, 248.08)</td>
</tr>
<tr>
<td>Number of mutants killed</td>
<td>126.35</td>
<td>(124.06, 128.63)</td>
</tr>
</tbody>
</table>

We had previously determined that setting $n = 10$, $s = 10$, $m = 500$ usually achieved the maximum coverage possible in doing randomized testing with the RUTE-J text interface. We therefore generated 50 minimal covering test suites using those parameters, and calculated how long it took to do so, the number of test cases and total number of method calls performed by them, and the number of lines actually covered.

Finally, we ran the test suites on the set of non-timeout mutants described in the last section, and calculated how many mutants each test suite killed.

5.2 Results

See Table 1 for a summary of the averages obtained from the experiment. For each average, we report on the 95% confidence interval calculated from the data.

We found that we could generate a minimal covering test suite in an average of 18.98 CPU seconds. On average, the total number of test cases in each suite was 46.12, with an average of 158.16 method calls in each suite, or 3.4 method calls per test case. As might be expected, the first few test cases in each test suite consisted of a single method call, each to a different method, since even one method call covers new code at first. Later test cases executed more complex situations like finding the last key in a tree that has more than one key, rebalancing on the left, and rebalancing on the right. The last ten or so test cases typically consisted of sequences of put and remove calls that executed small blocks of code in the rebalancing methods, representing unusual corner cases.

On average, the test suites covered 247.92 lines of code. Only one of the test suites did not cover the maximal 248 lines of code that the original randomized testing covered; the code left uncovered was one of the corner cases normally covered very late in the test suite generation process.

The test suites performed rather poorly at killing the TreeMap mutants; the average number killed was 126.35. Readers may recall that the original randomized testing reported on in the last section killed 182 non-timeout mutants, so the minimal covering test suites achieved an average of 69.4% test effectiveness on these mutants.

5.3 Discussion

The experiment corroborates previous research that suggests that simply achieving high coverage is not enough to achieve high test suite effectiveness (bug-finding ability). For instance, Rothermel et al. [30] found that for test suites that achieve 100% decision coverage, minimizing those test suites while maintaining decision coverage often resulted in reductions in effectiveness of 50% or more. Similarly, Hennessy and Power [18] found that minimizing test suites for grammar-based software while maintaining grammar coverage goals resulted in reductions of effectiveness of 96.5%, 59% and 100% for the three programs they studied.

Qualitative study of the results yields more insight. The mutants that were not killed by any of the generated test suites included all
those that failed to increment the size of the TreeMap on a put and all those that failed to decrement the size on a remove. The reason for this is that all test suites called the size() method in exactly one place, in an early test case containing only that call. The reason for this, in turn, is that the size() method simply returned the size field of the TreeMap, and thus it was possible to cover all lines of code in the method with one call.

While line coverage is a very weak form of coverage, we know of no standard structural coverage criterion that could be used instead of line coverage that would kill the problematic size() mutants. Even dataflow coverage criteria are usually defined over the code of a single method, but the size problem represents failures that can be revealed only by executing code in two methods in a specific order. Our conclusion is that no automatic test suite generation algorithm using standard coverage criteria as a goal will generate test suites that kill these mutants. Black-box, specification-based testing, or randomized testing, is needed for maximal effectiveness.

6. RELATED RESEARCH

Here we discuss other related research not mentioned earlier. The technique of randomized testing has long been known and has been applied to such diverse areas as compiler testing [26] and functional program testing [19]. Miller et al. [27] performed a classic study in which randomized inputs were given to Unix utilities, with an oracle limited to detecting crashes and hangs. There is also an extensive literature in statistical analysis of random testing, especially compared to category-partition testing [16, 13, 15]. This literature has identified the assumptions under which idealized models of randomized testing perform better or worse than other testing techniques, usually with respect to number of test cases.

Tillmann et al. [33, 32] have recently proposed "parameterized unit tests" that resemble the test fragments of RUTE-J in that they have parameters that can be instantiated with concrete values. However, their parameterized unit tests are single test cases that can be run like JUnit tests once the parameters are instantiated. RUTE-J test fragments, in contrast, are assembled randomly into sequences, each of which constitutes a new randomized test case.

The research that is closest in spirit to our own is recent research on the Tobias and Jartege tools [21, 29]. Tobias is given a schema of test cases and generates all instances of the schema, whereas Jartege is able to randomly generate test driver programs, each of which runs one test case. RUTE-J can be seen as a version of Jartege which wraps all potential test cases up in one execution, and adds useful features like a GUI and test case minimization.

7. CONCLUSIONS AND FUTURE WORK

RUTE-J is a Java package providing tool support to programmers for randomized unit testing. Using its GUI, programmers can find and minimize failing test cases that have been constructed randomly. In order to use RUTE-J, programmers write only Java code, in a form similar to JUnit test cases. Our experiments suggest that randomized unit testing with RUTE-J can be an effective, efficient method of testing, and that it can also be used as an experimental tool for studying other testing techniques.

There is a wealth of future directions to be explored. These include integrating coverage tools into RUTE-J and integrating RUTE-J into the Eclipse Java development platform, and adapting the RUTE technologies for other languages (for instance, RUTE-C). We would also like to do research on the relationship between randomized testing and model checking, by comparing the effectiveness and the development and execution costs of randomized testing, pure model checking, and randomized model checking.

8. ACKNOWLEDGMENTS

Thanks to Akbar Siami Namin for valuable comments and suggestions. This research is supported by a Discovery Grant from the Natural Sciences and Engineering Research Council (NSERC).

9. REFERENCES


