Comparing Automated Unit Testing Strategies

James H. Andrews[1], Yihao Zhang[2], and Alex Groce[3]

Report No. 736

December 2010

Department of Computer Science
University of Western Ontario
London, Canada
N6A 5B7

Abstract: Automated unit testing strategies that execute a large number of test cases are becoming more viable. In order to better understand the relationship between these strategies, we define "explorative" unit testing strategies as those which choose unit tests by exploring a large search space with a simple structure, and we study three particular explorative strategies: bounded-exhaustive, randomized, and a combined strategy.

In order to do this, we define canonical forms of unit tests and give precise definitions of the search spaces and strategies. We then show, by a combination of analysis and experimentation, that the bounded-exhaustive strategy is superior to randomized testing only when a small proportion of the search space of unit tests fails. We also show that we can arbitrarily and cost-effectively increase the failing proportion of the search space by simply increasing the number of method calls in the unit test.

[1]Department of Computer Science, University of Western Ontario; andrews@csd.uwo.ca
[2]Department of Computer Science, University of Western Ontario; yzhan694@csd.uwo.ca
[3]Department of Computer Science, Oregon State University; alex@eecs.oregonstate.edu

1 Introduction

Unit testing is the practice of testing methods, groups of methods or classes. A unit test is usually a piece of code written in the same language as the unit under test (UUT). The test code makes a sequence of method calls, each method call possibly preceded by code setting up the arguments of the call and possibly followed by code evaluating whether the methods did the right thing.

Recently it has become more cost-effective to run very large numbers of unit tests when testing critical units. Two trends have led to this development. The first is the trend toward reuse of general-purpose libraries; it is more important to assure the quality of heavily-reused units than of seldom-reused units. The second is the well-known trends of increasing processor speed and memory for the same price. To this we can now add the trend toward multi-core computing, since multiple test cases can be easily run on multiple processors.

In this paper, we examine a class of automated unit testing strategies that run large numbers of test cases. We refer to these as *explorative* strategies. Our definition of an explorative testing strategy is one in which we define a search space with a relatively simple structure, consisting of a very large number of test cases, and explore this search space systematically. In contrast to conventional black-box or white-box testing techniques, in which we devote most of our human effort to carefully choosing individual test cases that cover given criteria, in explorative strategies we concentrate our human effort on defining the search space, and we count on the processing power available to us to do the rest.

As an illustrative example, consider a hypothetical object-oriented UUT that has a single constructor taking no arguments, and three methods (m_1, m_2 and m_3) that each take two integer arguments. We define the search space as the space of all test cases consisting of a constructor call followed by 10 method calls. We restrict all the integer arguments to be in the range 0 to 9 because we believe that we can thoroughly test the unit even with the restriction in place. There are 300 possible method calls that can be done at any stage (3 choices of method, and 10 choices for each argument), so the number of possible test cases in the search space is 300^{10}, or approximately 6×10^{24}.

Two of the strategies that we study here are the *bounded-exhaustive* and *randomized* strategies. In the bounded-exhaustive strategy [7, 14], we systematically execute every test case in the search space; for the example unit, this would mean 6×10^{24} test executions. In the randomized-testing strategy [11], we instead choose test cases from the search space randomly, with the hope that if a failing test case exists, we will reach one relatively quickly; for instance, before executing 10^{10} test cases. We also study a strategy that combines the advantages of both bounded-exhaustive and randomized.

The word "bounded" in "bounded-exhaustive" refers to the bounds such as "10 method calls" and "integers between 0 and 9", which we choose in order to bound the search space. However, randomized strategies also rely on similar bounds. Typically, a randomized strategy does not try to avoid executing test cases that it has executed before, so randomized strategies are sometimes called "randomized testing with replacement", suggesting the metaphor of extracting a test case from the search space, running it and then putting it back in the search space.

There are two main potential benefits of explorative strategies like bounded-exhaustive and randomized. The first is that even the best white- and black-box testing strategies can miss failing test cases [10], whereas explorative strategies give the potential or the guarantee of executing any test case in the search space, including failing test cases that would not be predicted as failing by non-explorative strategies. The second is that the amount of human effort needed to define the search space may be less than the amount of human effort needed to define test cases that meet black- or white-box test adequacy criteria.

There are also two main potential problems for explorative strategies. The first is the "oracle problem": explorative strategies execute so many test cases that it's impossible for a human to evaluate them all to see if they succeeded or failed, requiring us to do automatic evaluation of test results. Typical approaches to the oracle problem include the definition of "high-pass" oracles, for instance that evaluate a test case as failing only if it leads to an uncaught exception [15]; defining oracles via formal specifications, for units with clear and simple formal specifications [8]; and deducing likely invariants by machine learning techniques [9].

The second potential problem for explorative strategies is the difficulty of optimally defining the search space of test cases. Antoy and Hamlet [4] and Doong and Frankl [8] may have been the first to observe that changes in parameters such as the length of test cases (number of method calls) and range of arguments can have a major impact on the effectiveness of the testing. Much recent work on explorative strategies concentrates on overcoming these potential problems, for instance by pruning the search space or by learning effective parameters [3, 15].

In this paper, we concentrate not on enhancing the benefits or ameliorating the problems of explorative strategies, though these are important areas of research. Instead we concentrate on comparing strategies directly to each other. In particular, we compare bounded-exhaustive (BE) unit testing, randomized (R) unit testing, and a combined strategy that we refer to as "Best of Both Worlds" (BOBW). In order to compare the strategies fairly and directly, we situate the work in the context of canonical-form unit test cases that we prove to be sufficiently general to encompass all explorative testing strategies. The main research contributions of this work are:

1. We show that any Java unit test case can be converted to each of several canonical forms.

2. We define the strategies BE, R and BOBW relative to the general notions of "test context" and canonical form.

3. We show analytically that, given reasonable assumptions, in the average case, the strategy R finds failing test cases more quickly than BE, except at low failure densities, and that BOBW finds failing test cases more quickly than either R or BE, at all failure densities.

4. We show analytically that in the average case, increasing the length (number of method calls) of a unit test case increases the failure density, increasing the viability of R compared to BE. We also show that this increase in length results in more failures per method call executed, making longer test cases more cost-effective, until a maximum cost-effectiveness is reached.

5. We give experimental evidence that, consistent with our analysis, the number of test cases needed to find failures in units is less with R than with BE, except at small test case lengths.

6. We show that, in our implementations of R, BE and BOBW, strategy BE takes longer in computation time to find a failing test case than strategy R.

The rest of this paper is organized as follows. In Section 2, we show that every unit test can be put into a canonical form. In Section 3, we define the strategies we study in terms of the search space of canonical-form unit tests. In Section 4, we compare the strategies by a mathematical analysis. In Section 5, we show by further analysis that increasing the length (in number of method calls) of a test case can cost-effectively cause randomized testing to be superior to bounded-exhaustive. In Section 6, we present an experiment that we performed with real subject units to corroborate our non-empirical analysis. In Section 7, we discuss the threats to the validity of the empirical results. In Section 8, we discuss the implications of the work.

2 Unit Test Canonical Forms

In this section, we show that every Java unit test has a *canonical form*: a simplified form into which it can be put which are equivalent to the original. The useful consequence of this is that as long as an explorative strategy can generate and run all canonical-form test cases, it can effectively perform any unit test case.

```
(a)
...
if (t.size() < n+1
   && !found) {
  x = t.get(n+42);
}
assert (x != 210);
```

```
(b)
...
int i1, i2;
i1 = t.size();
i2 = n+42;
x = t.get(i2);
b2 = (x != 210);
assert b2;
```

```
(c)
...
int i1, i2;
i1 = t.size();
i2 = 53;
x = t.get(i2);
b2 = false;
assert b2;
```

```
(d)
int[] intVP = new int[4];
intVP[0] = 53;
Tree[] treeVP = new Tree[1];
...
intVP[1] = treeVP[0].size();
intVP[2] = intVP[0];
intVP[3] =
  treeVP[0].get(intVP[2]);
booleanVP[1] = booleanVP[0];
assert booleanVP[1];
```

Figure 1: Canonical forms of unit tests. (a): Original unit test. (b), (c), (d): Test cases in canonical forms 1, 2 and 3 that are u-equivalent to (a), for some implementation of the units under test.

2.1 Definitions

We define a *Java unit test* as a sequence of Java statements which would compile correctly when given as the body of a method. We use the symbol T, possibly subscripted, to refer to an arbitrary Java unit test. We say that a unit test T *terminates unsuccessfully*, or *fails*, if it throws an uncaught exception, and that it *terminates successfully* or *succeeds* otherwise. (The use of the Java `assert` construct ensures that we can convert any Java unit test to such a form.)

We say that two Java unit tests T_1 and T_2 are *u-equivalent* if T_1 throws an uncaught exception at statement s if and only if T_2 does. In the Appendix, we actually define a sequence of three canonical forms; our main theorem about each canonical form is that, given a particular implementation of the methods that T_1 calls, there is a canonical-form test case T_2 which is u-equivalent to it.

Figure 1 shows an example of a Java unit test for a hypothetical Tree data structure, and some equivalent canonical forms. Our main focus is the rightmost canonical form, which is called Canonical Form 3 in the appendix.

2.2 Canonical Form 3

In a unit test in canonical form 3, all parameters for method calls are taken from "pools" of values stored in arrays. Canonical form 3 is particularly easy to generate automatically because, given some initial decisions, each of its statements can be generated by choosing a sequence of integers. The initial decisions are how big to make value pool variables, and what initial values to put into primitive-type value pools.

We define an *array-canonical method call* as an expression of one of the forms $m(\ldots)$, $new\ m(\ldots)$, $C.m(\ldots)$, or $e.m(\ldots)$, where m is a method name, C is a class name, and e and all the arguments of m are of the form $x[i]$, where x is a variable name and i is an integer constant.

We define an *array-canonical statement* recursively as follows. s is an array-canonical statement if either:

- It is an array-canonical method call;
- It is of the form $x[i] = e$, where x is an array variable name, i is an integer constant, and e is an array-canonical method call; or
- It is of the form `try { S } catch (E e) {x = e;}`, where `S` is an array-canonical statement.

We say that a Java unit test T is in *canonical form 3* if it is in four parts:

- A first part in which only array variables are declared and storage for them is allocated, where no more than one variable is declared of any given type. We refer to these variables as "value pools". For instance, the declaration `int[] intValuePool = new int[100]` declares a value pool for `int` of size 100.
- A second part in which constant values are assigned to elements of primitive type value pools; for instance, "`intValuePool[3] = 42`".
- A third part in which all statements are array-canonical statements.
- An assert statement of the form `assert` x, where x is a variable.

Theorem 1. *Let T be a Java unit test in which every variable which is used in an expression has previously been assigned a value, and whose arithmetic expressions do not throw exceptions. Then there is a Java unit test T' which is u-equivalent to T, and is in canonical form 3.*

Proof. See Appendix A. □

To summarize, every Java unit test case can be converted to a form which consists of a block of code setting up and initializing value pools; then a sequence of simple method calls that use the value pools as a source of target and parameter values, and a destination for return values; and finally an assertion.

We conclude that an automated testing strategy that can automatically generate all test cases of this form can generate a unit test that is u-equivalent to any failing unit test. This in turn means that if any test case can find a failure in a unit under test, then an automated testing strategy that can automatically generate test cases of this form can find a failure.

We will not consider finding value pool sizes and initial values in this paper, but note that given information about these choices, all the statements of a canonical form 3 unit test can be generated automatically. The statements in parts 1 and 2 can be generated deterministically, and the statements in part 3 can be generated by (a) choosing a method, (b) choosing a target for the method call if one is needed, (c) choosing values for the parameters, and (d) choosing a location to store the return value, if needed. The point of using value pools is that *every such choice is reduced to choosing an integer value pool index*. The `assert` statement in part 4 can be similarly generated by choosing an element from the boolean value pool.

We have therefore succeeded in reducing the problem of generating a general unit test to the three problems of choosing value pool sizes, choosing initial values, and generating a sequence of integers.

3 Definition of Strategies

In order to compare test strategies in a fair and precise manner, we must define those test strategies precisely, which we do here. Each of the strategies that we define is relative to a "test context", which is a structure describing choices that we make before beginning explorative testing.

In Subsection 3.1, we define the notion of test context. In Subsections 2 and 3, we show how a test context and a depth bound induces a finite search tree of test cases. Finally, in Subsection 4, we precisely define the three test strategies BE (Bounded Exhaustive), R (Randomized), and BoBW (Best of Both Worlds).

3.1 Test Context

Every explorative test strategy explores a search space within certain bounds. Only a certain set of methods are called, only a fixed number of different variables of a fixed set of types are declared, and the primitive-type parameters to the methods are chosen from a fixed set of values. We refer to these bounds as a "test context",

and formally define it here.

A *test context* consists of the following pieces of information:

- The set M_C of methods to be called. This includes all the methods actually under test, and may also include auxiliary methods that are needed to set up arguments for calls to the methods under test.
- The set T_I of types of interest. This should include all types (primitive types and classes) that are targets, parameters and return values of the methods to be called.
- For each type $t \in T_I$, an integer $vps(t)$ representing the value pool size for t. This is the number of separate values of that type that are available to act as parameters for a method call.
- For each primitive type $t \in T_I$, a mapping $init$ from indices i such that $0 \leq i < vps(t)$ to values of type t. This represents the initial values of the primitive type value pool elements.

Given a test context K, a test strategy has access to a *value pool* for each type $t_k \in T_I$. For each type t_k, we assume that value pool V_k is an array whose size is given by $vps(t_k)$.

Note that a choice of test context corresponds to parts 1 and 2 of a test case in canonical form 3.

3.2 Method Call Tuples

Given a test context $K = \langle M_C, T_I, vps, init \rangle$, and a depth (number of method calls) n, we can define precisely what test cases are encompassed by test strategies in that context.

We begin by defining a *parameter tuple* for a method or constructor m, which encodes the parameters to the call as a sequence of integers. We do this first in order to treat methods and constructors homogeneously, and methods homogeneously regardless of whether they are static or non-static, and whether their return type is void or non-void. In the following, V_k represents the value pool for type t_k.

- If m is a static method of class C with k parameters of types $t_1, \ldots, t_k$ and a void return type, a parameter tuple for m is a tuple of integers $\langle i_1, \ldots, i_k \rangle$, where each i_j is between 0 and $vps(t_j) - 1$ inclusive. The parameter tuple represents the call
  $C.m(V_1[i_1], \ldots, V_k[i_k])$.

- If m is a static method of class C with k parameters of types $t_1, \ldots, t_k$ and a non-void return type t_{k+1}, a parameter tuple for m is a tuple of integers $\langle i_1, \ldots, i_k, i_{k+1} \rangle$, where each i_j is between 0 and $vps(t_j) - 1$ inclusive. The parameter tuple represents the call $V_{k+1}[i_{k+1}] = C.m(V_1[i_1], \ldots, V_k[i_k])$.

- If m is a constructor of class t_{k+1} with k parameters of types $t_1, \ldots, t_k$, a parameter tuple is a tuple of integers $\langle i_1, \ldots, i_k, i_{k+1} \rangle$, where each i_j is between 0 and $vps(t_j) - 1$ inclusive. The parameter tuple represents the call $V_{k+1}[i_{k+1}] = new\ m(V_1[i_1], \ldots, V_k[i_k])$.

- If m is a non-static method of class t_{k+1} with k parameters of types $t_1, \ldots, t_k$, a target of class t_{k+1} and a void return type, a parameter tuple for m is a tuple of integers $\langle i_1, \ldots, i_k, i_{k+1} \rangle$, where each i_j is between 0 and $vps(t_j) - 1$ inclusive. The parameter tuple represents the call $V_{k+1}[i_{k+1}].m(V_1[i_1], \ldots, V_k[i_k])$.

- Finally, if m is a non-static method of class t_{k+1} with k parameters of types $t_1, \ldots, t_k$, a target of class t_{k+1} and a non-void return type t_{k+2}, a parameter tuple for m is a tuple of integers $\langle i_1, \ldots, i_k, i_{k+1}, i_{k+2} \rangle$, where each i_j is between 0 and $vps(t_j) - 1$ inclusive. The parameter tuple represents the call $V_{k+2}[i_{k+2}] = V_{k+1}[i_{k+1}].m(V_1[i_1], \ldots, V_k[i_k])$.

The definition of parameter tuple makes it clear that, within a test context and using the value pools defined by it, we can represent any parameter list as a sequence of integers: one integer representing the method, and the others representing the target, parameters and return value. Each parameter tuple corresponds to one of the statements in part 3 of a unit test case in canonical form 3.

In what follows, we will treat the target and return value of a method call, if any, as "virtual parameters" in positions $j = k + 1$ and $j = k + 2$.

3.3 Search Trees

We here define three classes of search trees for a given test context K: the parameter value search tree for a given method, the method call search tree for K, and the explorative strategy search tree for K. See Figure 2 for a diagram of these three classes of search trees.

Given a test context K, let the *parameter value search tree for method* m be constructed as follows: the tree has a root node at level 0, and a number of other levels equal to the number of parameters to the method. For $j >= 1$, where the jth parameter of m is of type t_j, every node at level $j - 1$ has $vps(t_j)$ children, representing the possible value pool locations from which to draw that parameter.

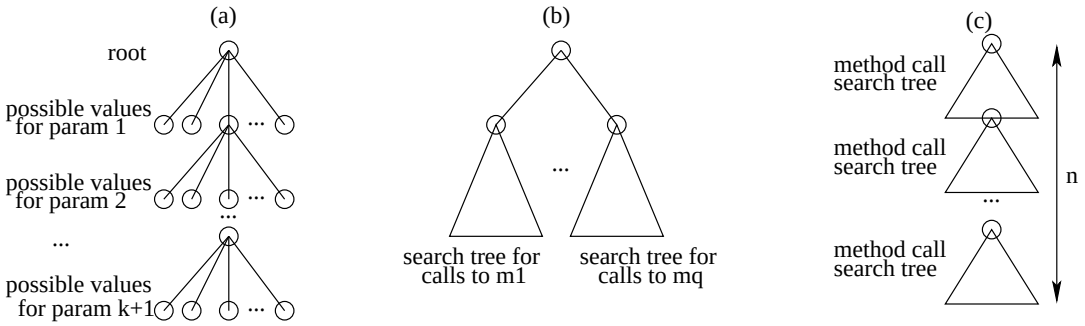

Figure 2: Search trees. (a): parameter value search tree. (b): method call search tree. (c): explorative strategy search tree.

A path from the root of this tree to any leaf of the parameter value search tree for m therefore encodes one method call tuple for m, as defined above. Note that the number of leaf nodes in the tree is the product of the value pool sizes of all the parameters (including virtual parameters).

Let the *method call search tree* for K be constructed as follows: the tree has a root node, and the root node has one child for each method m in the test context; that child is the root node of the parameter value search tree for method m. The number of leaf nodes in the method call search tree is the sum of the numbers of leaf nodes of all the search trees for calls to the methods m. In what follows, we will call this number j.

Finally, given a test context, we recursively define the *explorative strategy search tree for K for depth n*. The tree for depth 0 is the tree with just a single root node. The tree for depth n is constructed by constructing the tree for depth $n-1$, and then appending to each leaf node the method call search tree. Note that each path through the explorative strategy search tree, from root to leaf, records a unique sequence of n choices of method and, for each method chosen, the unique choice of parameters, target and return value location for the method call. There are therefore j^n leaf nodes in the explorative strategy search tree for depth n.

Given the discussion in Section 2, it should be clear that for every unit test case T, there is a test context and a value of n such that the explorative strategy search tree for depth n contains an encoding of the canonical form 3 version of T. The test context encodes the first two parts of the canonical form of T, and the explorative strategy search tree encodes the rest.

3.4 Test Strategies

We are now in a position to define the three strategies that we study in this paper: BE, R and BOBW.

We define the *(naive) bounded exhaustive test strategy for length* n, or BE(n), as the strategy that traverses the explorative strategy search tree in a depth-first fashion, executing the corresponding test case whenever it reaches a leaf.

We define the *randomized test strategy for length* n *and repetitions* q, or R(n, q), as the strategy that, q times, selects a random path from root to leaf of the explorative strategy search tree, and executes the corresponding test case. At every internal node, R selects the outgoing edge to follow using a uniform distribution. R is "randomized testing with replacement" because every time it selects a test case, it could be selecting one that it selected before.

Let the total number of leaf nodes in the explorative strategy search tree be z. We define the *best-of-both-worlds test strategy for length* n, or BOBW(n), as a strategy that explores the explorative strategy search tree by generating all the numbers from 0 to $z - 1$ in a pseudorandom order. After each number x is generated, BOBW chooses the test case represented by the path from the root to the xth leaf, and executes the corresponding test case. It therefore executes all of the test cases exactly once, but in a pseudorandom order *without* replacement. We give more details of the implementation of BOBW in Section 4.3.

Many published test strategies are variants or specializations of the first two strategies, and much research effort has gone into improving the strategies. For instance, Korat [14] performs isomorphism breaking to avoid executing essentially the same test case twice. This basic idea is also employed by Randoop [15], which generates short test cases randomly. Randomized testing is also the basis of the lower level of the Nighthawk tool [3].

Many of the published strategies also implement ways of finding test contexts in which given strategies perform well. While this is a crucial direction of research, in this paper, we separate the concerns of test context and search strategy in order to study in more depth the properties of search strategies themselves, independent of test context.

4 Analytical Comparison of Strategies

Explorative test strategies like BE or R execute all or some of the test cases in a large set of test cases, whose size is equal to j^n, the number of nodes in the explorative strategy search tree. We use the variable z to stand for j^n in this section. It is reasonable to compare BE with R; we do so here analytically, concluding that

BE is reliably superior to R only when failure density is low and failing test cases are not clustered.

4.1 Uniform Distribution of Failure

Let the number of failing test cases in the search tree be f. We define d, the *failure density* of the search tree[4], as f/z. The probability that k test cases randomly selected from the search tree are all non-failing is $(1-d)^k$. If we assume that failing test cases are spread evenly over the search tree, then clearly BE will find the first one in an expected number of test cases less than or equal to that of R, since R uses "randomized search with replacement".

However, it is likely that the failing test cases will not be spread evenly. This is because a fault in a method will lead to a failure only if that method is executed, and sometimes only if it is executed after given patterns of other method calls. The nodes in the search tree corresponding to failing test cases therefore tend to cluster in certain areas of the tree.

4.2 Non-Uniform Distribution of Failure

We start our analysis with a concrete example. Assume that BE finds its first failing test case after the first 0.1% of the sequence, i.e. after $z/1000$ test cases. We then expect R to do better than BE only if $(1-d)^{(z/1000)}$, the probability that R has not found a failing test case by that time, is less than $1/2$. This inequality is equivalent to

$$1 - d < e^{-(1000ln(2)/z)}$$

Since $e^x \geq 1 + x$ for any x, we expect R to do better than BE if

$$1 - d < 1 - (1000ln(2)/z)$$

that is, if f is greater than $1000ln(2) \cong 693.15$.

In general, if BE finds its first failing test case after z/p test cases, then R is expected to do better if there are $f = p * ln(2)$ failing test cases or more – a result that is *independent of the size z of the search tree*. Thus, as z grows, even if BE finds its first failure within the same fraction of the sequence, the failure density d above which R does better falls, favoring R more and more. For explorative unit testing strategies, z grows exponentially in the number n of method calls in each unit test.

[4]Note the distinction between *fault density*, which is typically measured in faults per 1000 lines of code, and *failure density*, which is a dimensionless quantity expressed as a ratio between number of failing test cases and total number of test cases.

Thus, except at low failure densities, R is likely to perform better than BE due to the risk of clustering of failing test cases. Because failing test case clustering is unpredictable, BE is inherently unreliable, better than R for some test contexts and worse for others. We therefore conclude that R is a better strategy than BE during development, initial testing and debugging, until failure densities are low enough that long random test runs find no failures and BE becomes more attractive.

4.3 Best of Both Worlds

The goal of the BOBW (Best of Both Worlds) strategy, which generates and runs all test cases in the search tree but in a pseudo-random order, is to gain the advantages of both R and BE. Because BOBW executes all test cases exactly once, like BE it avoids duplicate test case executions; but because it avoids depth-first traversal, it avoids the clustering that can defeat BE.

To generate the numbers, any linear congruential pseudorandom number generator with a full period will do [13]. For simplicity, our first implementations generated the next number in the sequence by adding a large prime number to the previous number and taking the remainder on division by z. (This is equivalent to a linear congruential pseudorandom number generator with the multiplier equal to the modulus.) A simple number-theoretic proof shows that this will generate all numbers from 0 to $z - 1$ before repeating.

We define the integer index of a test case in the following way. Assume a given test context K and a number n of method calls.

- Given a method m, let $t_1, t_2 \ldots, t_k$ be types of the k arguments to m (including the pseudo-arguments for target and return value). We define $nc(m)$, the number of distinct calls to m, as $vps(t_1) \cdot vps(t_2) \cdot \cdots \cdot vps(t_k)$, where $vps(t)$ is the size of the value pool for t.

- Let $(v_1, v_2, \ldots, v_k)$ be a sequence of k parameter values to m, such that $0 \leq v_i < vps(t_i)$ for all i. We define $[[(v_1, v_2, \ldots, v_k)]]$, the *index* of the parameter list $(v_1, v_2, \ldots, v_k)$, as the number

  $$v_1 + vps(t_1) \cdot (v_2 + vps(t_2) \cdot (\cdots (v_{k-1} + vps(t_{k-1}) \cdot (v_k)) \cdots))$$

  Given the index of a parameter list, it is possible to retrieve the parameters themselves by dividing by each of the $vps(t_i)$ in turn and taking the remainder as the value v_i.

- Let $m_1, \ldots, m_q$ be a listing of all the methods in M_I. We define j, the number of distinct method calls, as $nc(m_1) + \cdots + nc(m_q)$. We define

$[[m_p(v_1, v_2, \ldots, v_k)]]$, the *index* of the method call $[[m_p(v_1, v_2, \ldots, v_k)]]$, as the number

$$nc(m_1) + \cdots + nc(m_{p-1}) + [[(v_1, v_2, \ldots, v_k)]]$$

Given the index of a method call, it is possible to retrieve the method call itself by subtracting each of the $nc(m_i)$ in turn until the result is less than $nc(m_{i+1})$; this identifies which method is being called, and the parameters can be extracted as above.

- We define z, the number of possible test cases, as j^n, as noted earlier. Let $(mc_1, mc_2, \ldots, mc_n)$ be a test case, i.e. a sequence of method calls. We define $[[(mc_1, mc_2, \ldots, mc_n)]]$, the *index* of the test case $(mc_1, mc_2, \ldots, mc_n)$, as the number

  $$[[mc_1]] + [[mc_2]]j + [mc_3]]j^2 + \cdots + [[mc_n]]j^{n-1}$$

  Given the index of a test case, it is possible to retrieve the sequence of method calls by repeatedly dividing by j and taking the remainder as the index of the next method call.

The test case indices are so large that they typically cannot be represented in primitive-type numeric variables, but they can easily be represented in Java's standard `BigInteger` type. The representation takes a number of bits proportional to $\log(z)$, which is $n \log(j)$. The process of extracting the actual test case from its index takes n steps of length proportional to $\log(j)$. Note that the process of generating and running a test case for BE and R also takes time proportional to n.

5 The Effect of Increasing Test Case Length

We have shown that R is better than BE except at low failure densities. Here we show that there is a simple technique for increasing the failure density, namely to run longer test cases. We also show that not only do longer test cases yield higher failure density, but often yield more cost-effective testing.

5.1 Calculating Failure Density

Our previous work [2] suggests that feasible test case lengths for R can be as many as 100,000 method calls (depending on the cost of each method call).

Assume we have a test context K. Let the total number of distinct method calls that could be made at any step be j, and let our test cases be of length n, yielding j^n

possible test cases. Assume further that there is a failing sequence of method calls, of length k. We define $\Phi(j,k,n)$ as the number of test cases that fail, out of the j^n possible test cases. We can calculate $\Phi(j,k,n)$, but only under some assumptions. We will state these assumptions first, and then discuss the implications of dropping the assumptions. The assumptions are:

- There is only one failing sequence of method calls.
- There is no sequence u of method calls such that the failing sequence both starts and ends with u, unless u is the empty sequence or the whole failing sequence. This assumption holds, for instance, if the failing sequence starts with a constructor call which is not repeated elsewhere in the sequence.
- The failure can be detected whenever the failing sequence is in the test case. This is the case if, for instance, the unit under test is surrounded by a test wrapper that catches and processes expected exceptions and throws unexpected ones, which is a standard way of implementing high-pass test oracles used in explorative strategies.

Under these assumptions, we can calculate $\Phi(j,k,n)$, the number of failing test cases, as the following:

$$\Phi(j,k,n) = \begin{cases} 0 & \text{if } n < k \\ 1 & \text{if } n = k \\ \left(\begin{array}{c} j \cdot \Phi(j,k,n-1) \\ +j^{n-k} - \Phi(j,k,n-k) \end{array} \right) & \text{if } n > k \end{cases}$$

The first term in the third case of this equation (i.e., $j \cdot \Phi(j,k,n-1)$) represents the number of test cases of length n that have the failing sequence within the first $n-1$ method calls. The second term (i.e., j^{n-k}) represents the number of test cases that have the failing sequence at the end, i.e. that start with $n-k$ arbitrary method calls and end with the unique failing sequence. However, some of the latter test cases already have the failing sequence within the first $n-k$ method calls, so we have to subtract the number of such test cases, yielding the third term (i.e., $-\Phi(j,k,n-k)$).

$\Phi(j,k,n)$ therefore corresponds to a recurrence relation. The computer algebra tool Maple does not yield a simpler equation for this recurrence relation, but we implemented an infinite-precision calculator for it in Java using the standard `BigDecimal` class.

Given values of j, k and n, we define the *failure density* FD of the search space as the fraction of test cases that contain the failing sequence. This can be calculated as $FD(j,k,n) = \Phi(j,k,n)/(j^n)$. It is clear that for a given j and k,

as n increases, the failure density approaches 1. This is because as the test case increases in length, the probability of it containing the failing sequence approaches certainty (analogously, the probability of a given finite sequence of digits being somewhere in an infinite random sequence of digits is 1).

5.2 Calculating Failure Density Per Method Call

The previous subsection shows that we can arbitrarily increase the failure density of a testing task by increasing the length of the test cases. However, this is counterbalanced by the fact that running a long test case is expensive. The more important question is how *cost-effective* different test case lengths are.

For instance, given n, we can ask: is it more cost-effective to run the randomized testing strategy R($n, 2$), with 2 test cases of length n, or strategy R($2n, 1$), with one test case of length $2n$? If the failure density at length $2n$ is more than twice that at length n, then R($2n, 1$) will be more cost-effective. (The same reasoning applies to BE, since the failure density of the search space is the same.) We explored related issues empirically in an earlier publication [2]. Here we offer a computational analysis.

Given j, k, and n, we define the *failure density per method call*, or $FDpmc(j, k, n)$, as $FD(j, k, n)/n$. Given j and k, we would like to choose n so that $FDpmc(j, k, n)$ is as high as possible, because this yields a higher probability that each additional method call executed will fail. We wrote a Java program using the standard Java class `BigDecimal` in order to calculate $FDpmc(j, k, n)$ for many different values of j, k, and n. By increasing n and noting when the value began to decline, we were able to find the value of n having the first local maximum $FDpmc$ – that is, the point at which explorative testing is more cost-effective than for all previous values of n, but not more cost-effective than for the next value.

We found that for all but the shortest failing test sequences (1 method call), the optimal test case length n is greater than the number of possible method calls j, and the optimal length increases dramatically as k increases.

Note that these are reasonable values for k and n, but very low values for j; for more realistic values of j, the most cost-effective value of n is almost always greater than 1000. This in turn indicates that for all but the most trivial units under test and the most trivial faults (resulting in failing sequences of length 1), it is always more cost-effective to run test cases of length 1000 or more.

5.3 Dropping the Assumptions

Recall our assumptions:

- There is only one failing sequence of method calls.
- There is no sequence u of method calls such that the failing sequence both starts and ends with u, unless u is the empty sequence or the whole failing sequence.
- The failure can be detected whenever the failing sequence is in the test case.

If there is not only one failing sequence of method calls, then the failure densities will increase, but in a way that is more difficult to calculate.

However, it should be noted that even if there is only one failing sequence of length k, as test case length increases, it will be common for there to be more failing sequences of length $k+1$ and more. This is because there are often method calls such that, if they are inserted into the failing sequence, will not affect the failure of the rest of the sequence. We expect that this effect will tend to increase the failure density at length n beyond our calculated value of $FD(j,k,n)$.

Most failing sequences will start with one or more constructor calls which are never repeated through the rest of the sequence, since a failure will generally happen when a constructed object has been subjected to a series of operations that have changed its internal state. However, it is possible for a failing sequence to start instead with a call to a static method, or for constructor code to interact with other objects in such a way as to produce a subsequence that both starts and ends the failing sequence. In this case, the failure density calculated by $FD(j,k,n)$ will be an overestimate. We do not expect this situation to occur very often.

If failure is equated to throwing an uncaught exception, as in Section 2, then any execution of the failing sequence will result in detection of the failure. It is not always the case that failure is detected in this way, however; in some situations, the oracle (test result evaluation) is expensive and is deferred until the end of the test case. In such situations, the effects of later method calls can mask failures, so failure densities will not be as high in long sequences as our calculation of $FD(j,k,n)$ predicts.

In summary, it is not clear whether, in real-world units, the increase in failure density caused by dropping the first assumption is balanced out by a decrease in failure density caused by dropping the second and third assumption. It is therefore important to consider our calculation of $FD(j,k,n)$ as a simplified estimate, and to collect empirical data that will confirm or deny the trends that it shows.

6 Experimental Comparison of Strategies

In order to ground the above theory in empirical study, we implemented BE, R, and BOBW, and we ran experiments to compare their performance directly on real

Unit	SLOC	Mutants Compiled	Mutants Non-Equiv.
ArrayList	150	100	47
EnumMap	239	100	0
HashMap	360	100	29
HashSet	46	41	6
Hashtable	355	100	45
IHashMap	392	100	30
LHashMap	103	74	5
LHashSet	9	0	0
LinkedList	227	100	44
PQueue	203	100	38
Properties	249	100	1
Stack	17	33	28
TreeMap	562	100	24
TreeSet	62	45	8
Vector	200	100	92
WHashMap	338	100	38
Total	3512	1293	435

Figure 3: Data concerning experimental subjects.

units, and to measure the failure density of those units.

6.1 Subject Units

Our experimental subject was a set of heavily-used data structure units: the 16 units in *java.util* version 1.5 which inherit from the `Collection` and `Map` interfaces. These subjects contain a total of 3512 SLOC (lines of code not counting comments or whitespace).

6.2 Experimental Preparation

We generated mutants of the source files to act as faulty versions. Previous studies [1] have indicated that mutants can be good stand-ins for actual faults when assessing the effectiveness of testing techniques. We generated them using the same mutant generator as in [1], which generates mutants based on four classes of changes: “replace operator”, “replace constant”, “negate decision” and “delete statement”.

In order to simplify the experimental infrastructure, for each of the `java.util` classes, we also generated a "wrapper" class that instantiated the generic type parameters to `Integer`. Each wrapper class contained the same set of methods as the corresponding `java.util` class, but with the generic type parameters and the corresponding method parameters instantiated to `Integer`. For each `java.util` class, we selected as M_C (the methods to be called) all the methods in the class, and as T_I (the types of interest) all the types of all the method parameters, targets and return values.

We implemented the BE and R algorithms in Java. Each algorithm took as parameters a depth bound and two objects representing the test context. One of the text context objects referred to the original, "gold" implementation of the class and its methods. The other referred to a mutant implementation.

The test context that we used was one in which all primitive type value pools had two members and all class value pools had one member, and in which the primitive type value pools were initialized with distinct constants (e.g., 0 and 100 for the integer value pool).

For detecting the failure of a mutant, we implemented an approximation of what a test engineer with access to a good oracle would implement. Each algorithm generated and ran test cases on both the gold and the mutant version. Any exceptions thrown as a result of the method calls were stored in a list. At the end of the run of both test cases, the size of the exception list and the values in the primitive-type value pools were compared directly. If the size of the exception list was different or the values in the value pools were different, we judged that we had found a failure in the mutant unit (killed the mutant).

6.3 Experimental Procedure

The experiment proceeded in two phases. In the first phase, we identified which mutants were equivalent and which were non-equivalent. In the second phase, we measured failure densities and compared the strategies on the non-equivalent mutants.

For identifying which mutants were equivalent, we first ran strategy R(10, 1000), then R(100, 1000), then R(1000, 1000); that is, 1000 test cases of length 10, 100, and 1000. We did this first because we believed that R would be the best way to quickly identify failing test cases. In order not to bias our experiment in favour of R, we also ran BE testing with 3, 4, and 5 method calls per test case, until we either detected a failure or 30 minutes of clock time had passed.

For comparing strategies and measuring failure density, we first ran R(n, 1000), starting with $n = 1$ and increasing by 1 until $n = 8$, and then doubling n until $n = 1024$. On each run, we recorded how long R took to find its first failure (in

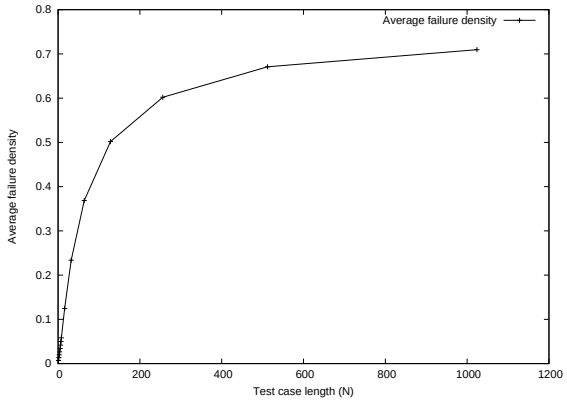

Figure 4: Failure densities for `java.util` mutants, by test case length.

CPU time and number of test cases), and how many of the test cases failed. We use $E(n)$ to stand for the index of the earliest failure at length n.

Running BE for a complete run for all but the shortest lengths was infeasible. We therefore ran BE(n) for $n = 1$ to 8, stopping as soon as a failure was found or $E(n)$ test cases were run. We collected information on whether a failure was found by BE, how many test cases were run, and how much total CPU time was needed.

6.4 Results

In the first phase of the experiment (identifying equivalent mutants), we found that 435 mutants over all `java.util` classes were non-equivalent; that is, that either R or BE could find a failing test case for 435 of the mutants. 434 of them were found by runs of R; only one (a mutant of `Hashtable`) was found by BE but not by R. This mutant was one which changed the order of entries in the hash table, causing its `toString` method to return a different string from the gold version.

In the second phase, the data we extracted allowed us to measure failure density. Figure 4 shows the failure density for the `java.util` units, averaged over all non-equivalent mutants of all subject units, as computed from the data from the runs of R. Consistent with our analysis, the failure density climbs steadily as n increases.

In order to examine whether the clustering of failing test cases mentioned in Section 4 occurs in practice, we examined the situations in which R($n, 1000$) could kill a mutant (i.e., find a failing test case for the mutant) and BE(n) could kill the mutant in fewer test cases. If failures are evenly spread throughout the search space, or clustered in a way that favours BE, we would expect that BE would kill

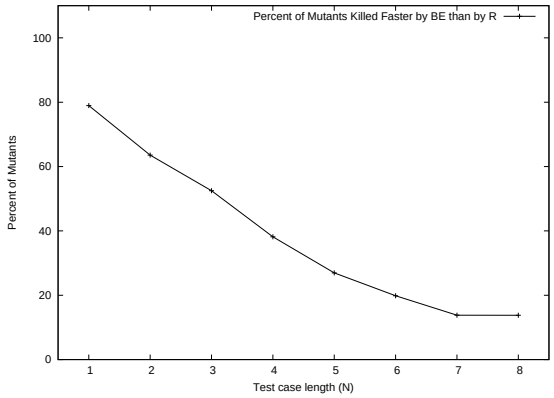

Figure 5: Percentage of cases in which BE(n) killed mutants in fewer test cases than R(n), for cases in which R could kill a mutant in fewer than 1000 test cases.

50% or more of the mutants more quickly than R. BE has the further advantage of not repeating test cases, which should give it the edge when failure densities are low, as explained in Section 4. Only if failures are clustered to BE's disadvantage should we see BE taking a larger number of test cases to kill mutants over 50% of the time.

Figure 5 shows the results of this comparison. BE indeed sometimes kills over 50% of the mutants in fewer test cases than R. However, this occurs only for short test case lengths ($n = 1$, 2, and 3), which as explained in Section 4 is where we expect the lowest failure densities. At $n = 4$ and higher, BE does not break even, indicating that the combination of higher failure density and failure clustering has rendered R more effective[5].

We also studied the total amount of CPU time taken by the runs performed in Phase 2. The statistic of interest here is number of failures found per CPU second. R achieved its lowest number of failures per CPU second (3.70) at $n = 1$. By $n = 8$, where the comparison to BE ended, it was achieving 15.44 failures per CPU second. In contrast, BE achieved its highest number of failures per CPU second (0.0014) at $n = 2$, and decreased consistently to a low of 0.00018 at $n = 8$. The poor performance of BE here is despite the fact that it ran a strictly smaller number of test cases than R. We should note, however, that optimizations such as symmetry breaking [14] would erase this advantage if they can make BE 2000 or

[5]Note that this graph must be interpreted carefully: it does not show that BE finds fewer failures than R, since we have restricted ourselves to mutants for which R finds failures in 1000 test cases or fewer. It therefore does not contradict the fact that a full run of BE for a given test case length will find failures that R will not find when running the same number of test cases.

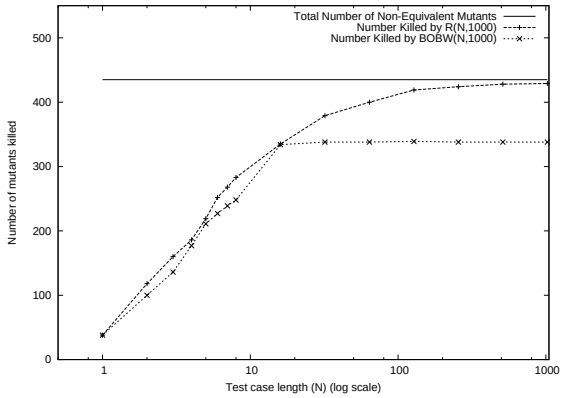

Figure 6: Number of mutants killed by R and BE (log scale x axis). The solid line is the number of mutants overall, i.e. the maximum number of mutants that could be killed.

more times faster.

Finally, we compared the number of mutants killed by our implementations of the R and BOBW strategies. The results are in Figure 6. It is clear that our BOBW implementation usually does not do as well as R, and in fact virtually "flatlines" after a test case length of 8.

Our interpretation of these results is as follows. We implemented BOBW by choosing a large prime number p and generating the next test case index by adding p modulo z. Given z, our implementation will result in a full period (all test indices from 0 to $z-1$ eventually chosen); however, if z is close to a multiple of p or vice versa, then there will be a (possibly small) finite number of points in the search space sampled, and then the next test cases will be very close to the previous ones.

Furthermore, when z is much larger than p, as happens very quickly when n increases, we explore only the initial part of the search space. The "flatlining" is due to the fact that the index generated determines first the initial method calls in the sequence; when n is large enough, the set of test cases executed for length n are identical to those generated for $n-1$, except that an extra index-0 method call is tacked on the end of each one.

Clearly, the choice of constants in our implementation of BOBW did not achieve the desired properties of a pseudorandom number generator. We are exploring other ways of implementing it.

7 Threats to Validity

We here summarize the threats to the validity of our experimental procedures.

Threats to internal validity would come only from bad development procedure or mistakes in data collection. We have guarded against them as much as possible, and have excluded from this paper any data that we are still reviewing for correctness.

Threats to construct validity would come from using inaccurate measures. We use mutant-killing ability as a measure of testing effectiveness; this is an increasingly-used practice but not without detractors. We have detected failures of the units under test by comparison to the gold version of the unit; this obviously does not reflect industrial practice, but is used here because we believe that the gold versions of the units that we studied are reliable. Also, we chose one particular test context to run, one with relatively small value pools. We can think of no reason why this would bias the experiment in favour of one particular testing strategy, but the possibility exists.

Threats to external validity would come from not drawing samples from a representative enough set of units. We indeed wish to run the experiments on a larger sample of units, in more languages. However, the `java.util` classes are a realistic, widely-used set of subject units which are often used in experiments. We believe that they are sufficient for the purpose of corroborating the results that we obtained first by analysis.

8 Discussion and Related Work

8.1 BE or R?

Our analysis and experiments here do not resolve the question of whether some particular, optimized implementation of BE would perform better or worse than some particular, optimized implementation of R on a particular unit under test. They attempt instead to abstract away from particular tools and implementations and study the mathematical structures underlying the questions. However, we can make some general observations.

We have shown that (naive) BE performs better than R (with replacement) when failure densities are low, and/or when failures are spread evenly over the whole search tree. The only possible source of this better performance is the re-running of previously used test cases by R, since the two strategies execute test cases from exactly the same search space.

Eliminating this re-running by using the BOBW strategy of generating all test case indices exactly once in a pseudo-random order will eliminate any difference

between the two strategies, unless other optimizations performed by some implementation are subverted by BOBW (we are unaware of any such optimizations). We can view the improvements brought by BOBW as either making BE less naive, or eliminating replacement from R; the effect is the same.

For very large search spaces, such as those that result from adopting large value pools and long test runs, it may not be realistic to perform a complete run of BE or BOBW (i.e., the running of all test cases), because the run would simply take too long. However, as the number of test cases run falls to a smaller and smaller fraction of z, R executes fewer and fewer duplicate test cases, and it becomes more and more likely that the overhead of BOBW (the large-integer arithmetic operations) will outweigh the benefit over R. Since BOBW will on average outperform BE in number of test cases run, this implies that in these situations R is the best strategy.

8.2 Related Work

Boyapati et al.'s system Korat performed BE testing [6, 14], where the bound ("scope") was defined as the size of input data structures accepted. Here we make a more general definition (length of sequence of method calls), and consider not only data structure testing, but unit testing in general. Boyapati et al. compared BE testing to random testing, finding that random search was usually not better than BE search. However, the depth bound for BE that they used in their experiments was just large enough to kill all mutants of the data structure code, and the depth bound for R was just one greater. This may have led to a situation in which the failure density was low, the situation in which BE performs better than R. As we show in this paper, the failure density can be increased simply by running longer sequences, and this increase in failure density is also cost-effective.

Coppit et al. [7] also studied BE testing, in the context of a case study of applying the tool TestEra to a complex fault-tree analysis tool. The authors concluded that BE was not able to generate inputs to meaningful bounds without refactoring the specification, but that when this refactoring was performed, BE testing found non-trivial faults.

Visser et al. found random testing competitive (in terms of coverage, execution time, and memory used) with model checking methods that in practice performed similarly to BE, and with various variations with and without state matching, symbolic execution, and abstraction of states [17, 16]. Explicit-state model checking with Java Pathfinder or SPIN provides an alternative explorative method, not considered in this paper because it is considerably more difficult to apply to many programs than BE or RT. Holzmann et al. give a survey of the most recent advances in this area [12].

In our own previous work [2], we found that the choice of test length did impact

the effectiveness of random testing, and that the number of failures per method call did rise to a maximum at long test case lengths. This work in part motivated the present paper.

The work that comes closest to implementing the BOBW strategy described here is the work by Pacheco et al. on the Randoop system [15]. Pacheco et al. generate random test cases, and avoid duplicate test cases by comparing the state of previously-run test cases. However, Randoop's strategy is optimized for generating short test cases, rather than the long test cases that we show in this paper are more cost-effective. Furthermore, instead of generating test cases that are guaranteed to not duplicate earlier ones, Randoop generates new test cases and then checks to see if they have been executed before; this will lead to more and more discarded test cases as the run proceeds.

A persistent problem in the research about random testing is the inconsistent definition of random testing. Arcuri et al., for instance [5], point out that many previous studies comparing a particular testing technique to random testing adopt a definition of random testing that inherently biases experiments against it. Our approach is similar in that it attempts to define randomized testing precisely, in order to compare it more precisely to competing approaches.

9 Conclusions and Future Work

We have shown, through a mixture of analytical and empirical methods, that randomized testing finds failures in less time and with a smaller number of test cases than naively-implemented bounded-exhaustive testing, unless failure densities are low. We have also shown that failure densities can be increased, partly negating the advantage of bounded-exhaustive, by increasing test case lengths. However, we have also shown that explorative testing can be implemented in a way that combines the advantages of both random and bounded-exhaustive strategies. These results help to clarify more precisely how, when and why randomized strategies can be useful in unit testing, and thus they may be useful for people implementing model checkers and other testing tools having an element of randomness.

In order to show the above, we defined a notion of equivalence of unit tests, and showed that all Java unit test cases can be transformed to an equivalent canonical form that can be generated easily. This led to a homogeneous definition of explorative test strategy that was used as the basis of the analytical and experimental comparison. These theoretical foundations may be useful in other contexts.

We are currently in the midst of extending the experiments reported on here to get more complete data comparing the three strategies proposed.

10 Acknowledgments

This work is supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). Special thanks to Leslie Goldberg for suggesting what she described as the "obvious" encoding of test cases as integers, which we used in the definition of BOBW.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005. 402-411.

[2] J. H. Andrews, A. Groce, M. Weston, and R. Xu. Random test run length and effectiveness. In *Proceedings of the IEEE/ACM Conference on Automated Software Engineering (ASE)*, pages Pages: 19–28, L'Aquila, Italy, September 2008.

[3] J. H. Andrews, F. C. H. Li, and T. Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *IEEE ASE'07*, 2007.

[4] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[5] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ACM International Conference on Software Testing and Analysis (ISSTA)*, pages 219–230, 2010.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 123–133, Rome, Italy, July 2002.

[7] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. J. Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–339, April 2005.

[8] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[10] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–11, December 1990.

[11] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[12] G. J. Holzmann, R. Joshi, and A. Groce. Model driven code checking. *Automated Software Engineering*, 15(3-4):283–297, 2008.

[13] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley, Boston, MA, USA, 1997.

[14] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT Computer Science and Artificial Intelligence Laboratory, September 2003.

[15] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 75–84, Minneapolis, MN, May 2007.

[16] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[17] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 37–48, Portland, Maine, July 2006.

A Proof of Canonical Form Theorems

In this Appendix, we prove that every Java unit test case can be converted to canonical forms 1, 2 and 3.

A.1 Assumptions and Scope

We make the following initial assumptions in order to simplify the proofs. For each assumption, there are well-known or straightforward transformations on test case or UUT code that would transform it into code conforming with the assumption.

- T contains no direct references or assignments to object fields with field access expressions such as `x.f`; instead, it accesses object fields using getter and setter methods, which are correctly implemented.
- T contains no references to inner classes inside other classes.
- Each case in a `switch` statement is terminated by a `break` statement, and there are no other `break` statements in T.
- There are no `continue` statements in T.
- T is not recursive, i.e. the test code is not in a method that ultimately calls itself.
- Every `if` statement in T contains an `else` clause.
- Every `if` clause, `else` clause, `while` body, etc., is enclosed by braces `{...}`, even if the clause or body contains a single statement.
- There are no compound-assignment statements in T, i.e. statements involving operators like "`+=`".
- There are no variable assignment expressions in T, i.e. instances of `(x = e)` embedded in other expressions.
- There are no increment ("`++`") or decrement ("`--`") operators in T.
- There are no anonymous class objects created in T.
- There are no local class declarations inside T.
- The arithmetic operations that appear in T do not themselves cause exceptions or errors to be thrown. Exceptions and errors can be thrown in called methods, and assertions in T can cause `AssertionError`s to be thrown, but the code of T does not do such things as dividing by zero.

We restrict the scope of our claims for simplicity, but we believe that our model is sufficiently general to be useful.

- We do not consider the fact that an object of a subclass can be passed to a superclass parameter.
- We assume T contains no `finally` clauses in `try` statements.
- We assume T does not raise out-of-memory conditions arising from a combination of test case code heap usage and UUT heap usage.
- We assume T does not contain the keywords `this` or `super`.

Note that `finally`, `this` and `super` can appear in the unit under test, just not in the test case code itself. Subclass parameters can be accounted for, with some loss of simplicity, by extending superclass value pools with elements that contain copies of subclass value pool elements.

A.2 Canonical Form 1

Intuitively, to convert a unit test to canonical form 1, we break all complex expressions out of their enclosing expressions or statements whenever possible, and simplify conditional statements according to how they are executed in the actual run of the unit test.

We define a *simple expression* as a variable or a constant. We say that a Java unit test T is in *canonical form 1* if it has the following properties:

- Every expression in T is either a simple expression, a unary operator applied to a simple expression, a binary arithmetic operator applied to two simple expressions, or a method or constructor call whose arguments are all variables. (For simplicity, we here consider an array element selection expression of the form $x[e_1, \ldots, e_k]$ to be a pseudo-method call $x.elementAt(e_1, \ldots, e_k)$.)
- Every statement in T is either of the form $x = e$, where x is a variable and e is a simple expression, or of the form `assert` e where e is a simple expression, or a method call whose arguments are all variables, or a `try...catch` block in which both the `try` block and the `catch` block consist of these kinds of statements. (For simplicity, we here consider an array element assignment statement of the form $x[e_1, \ldots, e_k] = e$ to be a pseudo-method call $x.set(e_1, \ldots, e_k, e)$.)
- Every declaration in T is of the form $type\ var$, i.e. it contains no assignment.

Theorem 2. *Let T be a Java unit test. Then there is a Java unit test T' which is u-equivalent to T, and is in canonical form 1.*

Proof. We state and (for non-trivial proofs) prove the following propositions about u-equivalence. Each proposition about a statement S_1 being equivalent to S_2 induces a transformation from a unit test containing S_1 to the corresponding unit test with S_1 replaced by S_2. We can therefore view these propositions equivalently as steps in a transformation process.

1. Let s be a statement in T of the form "`while (e) { S }`", and let s be not enclosed within any other loop construct. Then s is u-equivalent to "`boolean b; b = e;`", where `b` is a new variable (variable not appearing in T), followed by zero or more repetitions of "`S; b = e;`".

   Proof: since s is not enclosed in any other loop, s as a whole will be executed 0 or 1 time (depending on whether it is enclosed in a statement such as an `if`, `case` or `try...catch`, and depending on the branch of that statement taken). If it is not executed, then the new code acts the same as the old code. If the while loop is executed, then the loop contents will be executed n times, $n \geq 0$. It is therefore equivalent to an initial evaluation of the loop decision `e` followed by n repetitions of the loop contents `S`, each followed by a re-evaluation of the loop decision.

2. Let s be a statement in T which is some other looping construct (e.g., a `for` or `do...while` loop) not inside any other loop construct. Then s can be transformed in a similar way to a `while` loop.

   * After a finite number of repetitions of steps 1-2, we can assume that there are no loops in T. We will assume this from now on.

3. Let s be a statement of the form "`if (e) {S1} else {S2}`", where `e` is not a variable. Then s is u-equivalent to "`boolean b; b = e; if (b) {S1} else {S2}`".

4. Let s be a statement of the form "`switch (e) {...}`", where `e` is not a variable. Then s is u-equivalent to "`T x; x = e; switch (x) {...}`", where `T` is the appropriate type.

5. Let s be a statement of the form "`if (x) {S1} else {S2}`", where `x` is a variable. Then s is u-equivalent to either `S1` or `S2`.

   Proof: in the actual execution of the `if`, either one branch or the other will be taken.

6. Let s be a statement of the form "`switch (x) {...}`", where `x` is a variable. Then s is u-equivalent to one `case` inside the `switch`.

* After a finite number of repetitions of steps 3-6, we can assume that there are no `if`s or `switch`es. We will assume this from now on.

7. Let s be a statement of the form "`assert e;`", where `e` is not a variable. Then s is u-equivalent to "`boolean x; x=e; assert x;`".

8. Let s be a statement involving a method call at its top level, of one of the following forms:
   ```
   e.m1(...).m2(...);
   y = e.m1(...).m2(...);
   ```
   where `e` is an expression. That is, s involves a method call followed by another method call on the result of the first call. Then s is u-equivalent to the following (respectively), where `z` is a new variable and `T` is the appropriate type:
   ```
   T z; z = e.m1(...); z.m2(...);
   T z; z = e.m1(...); y = z.m2(...);
   ```

9. Let s be a statement of the form "`e;`" or "`x = e;`", where `e` is a method or constructor call of one of the following forms:
   ```
   m(x1, ..., xk, e', ..., xn)
   C.m(x1, ..., xk, e', ..., xn)
   new C(x1, ..., xk, e', ..., xn)
   y.m(x1, ..., xk, e', ..., xn)
   ```
   where each `x` parameter is a variable, and `e'` is not a variable. Then s is u-equivalent to "`T w; w = e';` s'", where s' is s with the occurrence of `e'` replaced by `w`.

10. Let s be a statement of the form "`x = e1 op e2;`", where `op` is the operator "`&&`" or "`||`". Then s is u-equivalent either to "`x = e1;`" or "`x = e1; x = e2;`".

    Proof: by Java's short-circuit (McCarthy) evaluation, if `op` is "`&&`" (resp. "`||`"), then evaluation will end if the left-hand operand evaluates to false (resp. true). If the left-hand operand evaluates to true (resp. false), then both operands will be evaluated, and the value of the whole expression will be the value of the right-hand operand.

11. Let s be a statement of one of the following forms:
    ```
    x = e1 op e2;
    x = op e1;
    ```
    where `e1` is not a variable, and `op` is an operator but not "`&&`" or "`||`". Then s is u-equivalent to the following respective form:

```
T y; y = e1; x = y op e2;
T y; y = e1; x = op y;
```

12. Let s be a statement of the form "`x = y op e2;`", where `x` and `y` are variables, `e2` is not a variable, and `op` is an operator but not "`&&`" or "`||`". Then s is u-equivalent to "`T z; z = e2; x = y op z;`".

13. Let s be a statement of the form "`x = (e1 ? e2 : e3);`". Then s is u-equivalent either to "`boolean y; y = e1; x = e2;`" or "`boolean y; y = e1; x = e3;`".

    * After a finite number of repetitions of steps 7-13, we can assume that all assignment statements in T are of the form "`x = e;`", "`x = op z;`", "`x = y op z;`", or "`x = m(y1, ..., yn);`" (or its class, non-static and constructor variants), where `x` and all operands are variables, and `e` is either a constant or a variable. We can also assume that all method call statements are of the form "`m(y1, ..., yn);`", "`C.m(y1, ..., yn);`", or "`x.m(y1, ..., yn);`", where all the `y`s are variables. We will assume this from now on.

14. Let s be a statement of the form "`try { S } catch (T1 x) { S1 } ... catch (Tn x) { Sn }`". Then s is u-equivalent to `S`, or to "`try { S } catch (Ti x) { Si }`", for some i.

    Proof: in the actual execution, zero or one of the `catch` blocks will be executed.

15. Let s be a statement of the form "`try { S1 ; S2 } catch (T x) { S3 }`". Then s is u-equivalent either to "`try { S1 } catch (T x) { S3 }`", or to "`S1; try { S2 } catch (T x) { S3 }`".

    Proof: If `S1 ; S2` raises a `Throwable`, then either it will do so in `S1` or in `S2`. If `S1` raises a `Throwable` which is an instance of type `T`, then `S2` will not be executed and `S3` will be executed. If `S1` does not raise a `Throwable` at all, then its execution will be identical to as if it were not in the `try` clause. If `S1` raises a `Throwable` which is not an instance of type `T`, then neither `S2` nor `S3` will be executed, which again is identical to the situation where it is not in the `try` block at all.

After a finite number of repetitions of all of the transformations induced by the above equivalences, the test case will be in canonical form 1. □

A.3 Canonical Form 2

It is reasonable to assume that a unit test T does not contain uses of variables that have not been assigned a value, and also that the arithmetic operators in T do not cause exceptions to be thrown. (The methods that T calls may do these things, but the code of T itself does not.) Under these assumptions, we can further simplify a unit test to canonical form 2, in which no arithmetic calculations are done.

We say that a Java unit test T is in *canonical form 2* if it is in canonical form 1, and in addition all assignment statements are of the form $x = e$, where e is either a constant or a method or constructor call.

Theorem 3. *Let T be a Java unit test in which every variable which is used in an expression has previously been assigned a value, and whose arithmetic expressions do not throw exceptions. Then there is a Java unit test T' which is u-equivalent to T, and is in canonical form 2.*

Proof. By the previous Theorem, we can convert T to a unit test T'' in canonical form 1. Every statement in T'' of the form $x = e$ where e is an arithmetic expression terminates successfully, yielding a deterministic value e', without any methods or constructors being called. We can therefore replace the statement by $x = e'$. Every statement in T'' of the form $x = y$ where y is a variable can be eliminated if later references to x are replaced by references to y until x receives another value. The result will be a unit test T' in canonical form 2. □

A.4 Canonical Form 3

Recall Theorem 1: *Let T be a Java unit test in which every variable which is used in an expression has previously been assigned a value, and whose arithmetic expressions do not throw exceptions. Then there is a Java unit test T' which is u-equivalent to T, and is in canonical form 3.*

Proof. By the previous Theorem, we can convert T to a unit test T'' in canonical form 2.

All assertions in T'' are of the form `assert` x, where x is a variable. We can eliminate all of the assertions for which x evaluates to true, and eliminate all of the statements following the first assertion where x evaluates to false.

For every type t in the unit test, we can count the number u_t of separate variables of that type, and the number w_t of constants of that type used in T''. Let the number $u_t + w_t$ be called the *value pool size* of t, abbreviated $vps(t)$.

For a given type t, let V_t be the name of the value pool for t. We can convert T'' to canonical form 3 by the following transformations.

- Replace each reference to v_i, the ith variable of type t (counting from 0) by the expression $V_t[i]$.
- Replace each reference to c_j, the jth constant of type t (counting from 0) by the expression $V_t[k]$, where k is $u_t + j$.
- Precede the transformed statements of T'' by one block of statements for each type t mentioned in T. The block of statements for a given type t has the form:
  $t[\,]\ V_t$;
  V_t = new $t[vps(t)]$;
  $V_t[u_t]$ = c_0;
  $V_t[u_{t+1}]$ = c_1;
  ...
  $V_t[u_t + w_{t-1}]$ = $c_{w_{t-1}}$;

The final unit test will be in canonical form 3. □