2011

COMPARING AUTOMATED UNIT TESTING STRATEGIES

Yihao Zhang

COMPARING AUTOMATED UNIT TESTING STRATEGIES

(Spine Title: Comparing Automated Unit Testing Strategies)

(Thesis Format: Monograph)

by

Yihao Zhang

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario
February, 2011

Abstract

Software testing plays a critical role in the software development lifecycle. Automated unit testing strategies allow a tester to execute a large number of test cases to detect faulty behaviours in a piece of software. Many different automated unit testing strategies can be applied to test a program. In order to better understand the relationship between these strategies, "explorative" strategies are defined as those which select unit tests by exploring a large search space with a relatively simple data structure. This thesis focuses on comparing three particular explorative strategies: bounded-exhaustive, randomized, and a combined strategy. In order to precisely compare these three strategies, a test program is developed to provide a universal framework for generating and executing test cases. The test program implements the three strategies as well. In addition, we perform several experiments on these three strategies using the test program. The experimental data is collected and analyzed to illustrate the relationship between these strategies.

Keywords: Software Testing, Unit Testing, Testing Strategies, Bounded Exhaustive Testing, Randomized Testing

Acknowledgments

Millions of thanks go to Dr. Jamie Andrews, who gave me the precious opportunity to perform researches with him. He is the lighthouse of my research and learning from him is one of the most valuable experiences in my life. Jamie has offered great help on the thesis topic, algorithms, application development and experiment setup and analysis. It is impossible to have finished this thesis without his help.

Sincere thanks to Dr. Eric Schost who reviewed my thesis proposal and suggested several refinements on it.

Many thanks to my friends in Canada and other countries, Jian Zhu, Yongmin Shuai and so forth. I cannot list all their names here but I am very grateful for their support and help.

The most appreciation and gratefulness go to Jiaying Shen. Her accompanying, support and inspirational words have stimulated me to finish this thesis and obtain many achievements.

Special thanks go to my family, Guoshun Zhang, Fengying Du, Guoqing Zhang, Xiaping Zhang, Yujie Zhang, Guoan Zhang, Shujun Chen, Shuting Zhang. The tremendous spiritual and financial support that they have given me has always been the light in the darkness.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

The software development lifecycle (SDLC) is the process of creating or modifying software systems and the models and methodologies which people use to develop software systems. The SDLC consists of the following main phases: system planning, requirements gathering and analysis, system design, implementation, testing, and maintenance. The overall quality of the software system heavily depends on the quality of the execution of each phase in the SDLC. This thesis concentrates on making improvements in the testing phase. We will discuss problems that software testers or software quality assurance personnel encounter in the course of testing a piece of software. These problems are major concerns to a type of software testing methodology called unit testing. We have developed solutions to these problems to figure out the relationship among different automated unit testing strategies.

1.2 Software Testing

Software testing plays a very important role in the software development life cycle. It is used to evaluate and ensure correctness, completeness and quality of a piece of software. Software testing also facilitates making any improvements which are deemed to be indispensable. Although it's impossible to ensure a software program is free of problems which are called bugs, we can create or adopt a well developed testing strategy that can increase possibilities of finding a fault if one exists.

Many different approaches can be applied to software testing [10]. Depending on the type of the software implementation, different testing approaches have different objectives and yield different results. Software testing traditionally can be divided into two categories using the box approach: black-box testing and white-box testing. Black-box testing treats the software program as a "black box" without any knowledge about its internal implementation. Black-box testing is to verifying the correctness of the functionality of the software, while white-box testing validates the correctness and completeness of the actual source code. System testing is a thorough testing of the entire software system while regression testing tries to ensure the correctness of the functionality of the existing software after new features have been integrated or bugs have been fixed. More importantly, unit testing refers to verify the functionality of a specific section of code. Unit testing lays a foundation for system testing since it checks for correctness of each small part of the software system that is included in the software package, confirming that they work correctly according to the specification when they run separately. After that, integration testing can be performed to test programs which modules are grouped together. Finally, based on the assumption that previous testing has eliminated all the bugs and all underlying modules work correctly, system testing can also be carried out to test the overall system.

1.3 Unit Testing and Automated Unit Testing

Unit testing is an approach in which an individual piece of code is tested to determine whether it works correctly and meets the specification. Unit tests are often created by programmers or sometimes by testers to conduct white-box testing. A unit is the smallest part of a software program to test. Unit testing is used to validate the correctness and completeness of a unit. Each unit is tested separately before integrating them into modules to test the interfaces between modules. As a result, it brings several benefits. One of them is to help software developers detect errors and defects as early as possible in software development life cycle.

There are several unit testing frameworks for various programming languages. For example, JUnit is a unit testing framework for the Java programming language. It provides many features to facilitate software developers writing unit test cases. A unit test case is often written manually by software developers. However, this process can be very tedious and time-consuming. In addition, it may not be effective in finding certain classes of problems. Therefore, test automation, especially unit test automation is necessary to accelerate the unit testing process. Once tests are automated, they can be run very quickly. This is often the most cost-effective way to test and maintain software products in the long run. Briefly speaking, automated unit testing is a unit testing process of writing a computer program to do the testing which otherwise needs to be done manually. There are two general approaches to automate tests:

- Code driven test automation. Methods, classes, packages, and modules are tested automatically with various input arguments to verify whether the return value is correct.
- User interface driven test automation. A testing program generates user inputs such as keyboard input and mouse clicks to observe changes in the user interface and validate that the observed behavior of changes is correct.

This thesis focuses on code driven test automation. Different automated unit testing strategies can be applied to test the software program. They often yield various results and abilities to detect errors. In the following subsections, this thesis will provide brief introductions for two automated unit testing strategies: bounded exhaustive unit testing and randomized unit testing.

1.3.1 Bounded Exhaustive Unit Testing

One of the unit testing strategy this thesis examines is *bounded exhaustive unit testing.* Bounded exhaustive unit testing is a unit testing technique in which software is automatically tested with all valid inputs until it reaches specific size bounds. Running a test case consists of executing a sequence of method calls in the subject unit we would like to test. For bounded exhaustive unit testing, it requires not only generating all valid values of input arguments for a method but also testing all possible sequences of method calls in the subject unit.

1.3.2 Randomized Unit Testing

Another unit testing strategy this thesis examines is *randomized unit testing.* Randomized unit testing is a unit testing technique in which software is automatically tested with randomly selected input arguments and method calls. Like bounded exhaustive unit testing, running a test case in randomized testing consists of executing a sequence of method calls. However, randomized unit testing requires randomization in selecting method calls and selecting input arguments to be passed into the particular chosen method call. When it is utilized properly, it has been found that randomized unit testing is efficient and easy to perform.

1.3.3 Best of Both Worlds

The last unit testing strategy this thesis examines is *best-of-both-worlds.* Best-of-both-worlds unit testing is a combined unit testing technique in which software is automatically tested with pseudo-randomly selected input arguments and method calls until it exhaustively takes all valid inputs within specific size bounds. Compared with bounded exhaustive testing, best-of-both-worlds takes all valid inputs within specific bounds but in a pseudo-random order. Compared with randomized testing, best-of-both-worlds generates test cases in a random order but unlike randomized testing, it doesn't select the same test case twice. Generally speaking, we would predict that it is a better strategy than either bounded exhaustive or randomized because it combines the advantages of both strategies.

1.4 Test Oracle

A test oracle is used to determine whether a piece of software behaves correctly after test execution. It is often used by software testers and developers to determine whether a test has passed or failed. For a given test input, a test oracle compares the output of the system under test with the output which a test oracle expects the system should have. Therefore, a test oracle should always be separated from the system under test in order to correctly verify the system. Based on the types of system under test, different test oracles can be applied to test. Baresi et al. [5] have surveyed several approaches to test oracles:

- Embedded Assertion Languages,
- Extrinsic Interface Contracts,
- Pure Specification Languages,
- Trace Checking,
- and Log File Analysis.

This thesis uses our own test oracles to verify the test output. The mechanism of our test oracle will be explained in detail in later chapters.

1.5 Thesis Focus

Figuring out the relationship among bounded exhaustive, randomized and best-of-both-worlds unit testing strategies is critical to software testing research because it helps us to identify which testing strategy should be adopted in unit testing in order to find software failures more effectively. In order to compare those three unit testing strategies precisely, Andrews et al. [4] introduce the canonical form of unit test cases that is proved to be sufficiently general to encompass the three testing strategies. Based on the proofs and canonical forms in [4], this thesis designs and implements several experiments to compare those unit testing strategies. The experimental results demonstrate the correctness and some assumptions described in [4].

First of all, we will discuss the design of the test program which implements the three unit testing strategies and runs test cases in Java programming language. One of the essential parts of this thesis is design decisions, architecture and implementations of the test program. The test program implements bounded exhaustive, randomized, and best-of-both-worlds strategies using different algorithms respectively. In addition, the test program needs to be flexible, adaptive to change and easy to maintain. A good design of the program plays a vital role in fulfilling those non-functional requirements.

Another essential part of the thesis is design and implementation of the experiments which compare the three unit testing strategies from different perspectives. The ultimate goal of the experiments is to compare bounded exhaustive, randomized and best-of-both-worlds unit testing strategies in two facets: abilities and effectiveness to find failing test cases. For each experiment, preparation, goal, procedure and collected data will be described in detail. Based on the data that have been collected during experiments, we will give several diagrams, plots and tables to illustrate the comparison among bounded exhaustive, randomized and best-of-both-worlds.

Thorough analysis of the collected data and experimental results is also indispensable. Judging by the experimental data, we will discuss the situations in which one strategy outperforms the others in time to first failure. Furthermore, based on the theoretical analysis in [4] and experimental data, the thesis concludes that increasing the number of method calls of a unit test case increases the failures distributed in the whole search space which also increases the viability of randomized compared to bounded exhaustive. Our experiments have shown that increasing the length of a test case (number of method calls) results in more failures per method call executed, which means making longer test cases more cost-effective, until a maximum cost-effectiveness is reached. Our research in this thesis demonstrates that on average, randomized unit testing strategy outperforms bounded exhaustive strategy in its effectiveness and ability to find failures in the subject units under test.

1.6 Thesis Organization

Introduction and other relevant background information have been highlighted in chapter 1. We will introduce some related work that has been done concerning automated unit testing and unit testing strategies in chapter 2. In chapter 2, some important concepts regarding our research will be explained as well. We will talk about design and implementation of the test program in chapter 3. In chapter 4, we will focus on the design and implementation of experiments in comparing bounded exhaustive, randomized and best-of-both-worlds unit testing strategies. Any problems and issues that occur during the experiments will be discussed as well. We will analyze experimental data, illustrate experimental results and draw some conclusions in chapter 5. In chapter 6, we will present some future research areas.

Chapter 2

Related Work

In this chapter, we give some basic definitions, and then discuss related work in bounded exhaustive testing, randomized testing, best-of-both-worlds, mutation and the most relevant work – Andrews et al.'s approach of comparing automated unit testing strategies.

2.1 Definitions

Here we define some terms that will be useful through the rest of this thesis.

An **explorative testing strategy** is a strategy in which we define a large search space with a relatively simple structure, consisting of a large number of test cases, and explore this search space systematically [4].

A **failing test case** is a test case which will cause the unit under test to fail. If the execution result of a test case is not what we expected, we can declare the test case to be a failing test case.

A **passing test case** is a test case which does not cause the unit under test to fail.

2.2 Bounded Exhaustive Testing

The idea of bounded exhaustive testing is first proposed by Marinov and Khurshid in [12]. This is a testing strategy which exhaustively tests all valid input up to a specific size or bound. In [12], Marinov and Khurshid proposed a novel framework named TestEra for automated specification-based testing of Java programs. Given a formal specification, TestEra uses the method precondition to automatically generate all inputs up to a given bound. It does not require user input besides a method specification and an integer bound with integer for input size. In [12], Marinov and Khurshid only analyzed test cases with small input bounds. TestEra may encounter performance issues when it is given large input bounds to test the program.

Many published testing strategies are variants or specializations of the bounded exhaustive testing strategy, and much research effort has gone into improving the strategies. For example, Marinov et al. [11] developed Korat, a testing framework which systematically enumerates all legal inputs within a certain size. Korat performs isomorphism breaking to avoid executing the same test case twice. Developers can provide a precondition predicate, written in a standard programming language, and Korat identifies whether the input satisifies the required invariants. Korat then processes the predicate to produce a stream of structures that satisfy the property identified by the pre-condition. To test the program, Korat generates all valid inputs within a certain bound which satisfies the invariants. It tests the program on the generated inputs to verify that the execution meets the provided post-condition. Marinov et al. use mutation testing to measure the quality of the test suites that Korat generates. Concepts and approaches of mutation testing will be further explained in this thesis. Marinov et al. conclude that bounded exhaustive testing (or the term exhaustive testing used in [11]) within some scope can be more effective than random testing with bigger inputs. However, the depth bound for bounded exhaustive testing that they used in their experiments was just large enough to kill all mutants of the data structure code, and the depth bound for randomized testing was just one greater. This may lead to a situation in which failures are mainly distributed in a low level of the whole search space so that bounded exhaustive testing is able to outperform randomized testing. Therefore Marinov et al.'s conclusion doesn't necessarily mean bounded exhaustive testing is a better testing strategy than randomized testing. More empirical studies are indispensable to compare bounded exhaustive testing and randomized testing.

Coppit et al. [7] also studied bounded exhaustive testing by applying TestEra to the Galileo dynamic fault tree analysis tool, a complex production software system. Coppit et al. empirically studied the feasibility and potential utility of bounded exhaustive testing. The authors concluded that bounded exhaustive testing has better bug-detecting abilities than manual ad-hoc testing in which a suite comprises at most a few hundred tests. However, the reliability of bounded exhaustive testing may be jeopardized by two aspects. The first aspect is errors in the test oracle and the second is the specification from which tests are generated. Additionally, Coppit et al. point out that it is always possible, in general, that a behavior just beyond the tested bound will be erroneous. This fundamental limitation lies behind every testing strategy as well as bounded exhaustive. Another significant problem with bounded exhaustive testing in [7] is that bounded exhaustive testing was not able to generate inputs to meaningful bounds without refactoring the specification. Selectively reverse engineering a specification from which both a characterization of well-formed inputs and an oracle are derived is the key element of applying bounded exhaustive testing. When refactoring the specification was performed, bounded exhaustive testing can be effective and feasible to reveal previously unknown bugs in the system under test.

2.3 Randomized Testing

Randomized testing or random testing is a simple testing strategy of generating randomized input and feeding it to the software under test. It is mentioned as early as Myers in 1979 [14]. Myers believed that in general, the least effective methodology of all is random-input testing which is the process of testing a program by selecting, at random, some subset of all possible input values [14]. Although Myers' book is cited by many research works on software testing, the judgement of randomized testing is biased due to lack of empirical studies.

Past research on randomized testing included that of Claessen and Hughes on QuickCheck [6]. QuickCheck is a testing tool that utilizes randomized testing to test Haskell programs. Using formal specifications, QuickCheck allows testers to define certain properties of the functions under test that should be expected and check whether the properties hold after running several test cases. The tool also is able to automatically generate test cases based on random inputs or based on custom defined test data generators.

Miller et al. [13] has also proven the effectiveness of random testing to end users and developers. By simply randomly generating strings of characters using a program called *fuzz*, they found that a surprisingly large number of UNIX utility programs either terminate abnormally, loop infinitely or terminate without a clear description of what has happened, totaling to more than 24% of the basic UNIX utility programs. Their research also pointed out several common mistakes made by programmers.

Randomized unit testing is a specific type of randomized testing which automates the testing by randomly selecting or generating sequences of method calls and inputs. Andrews [1] focused on coverage-checked random unit testing (CRUT), which applies randomized unit testing strategy to a unit under test, continuously testing it until predefined coverage criteria are achieved. Andrews concluded that CRUT is efficient in finding faults within the code and it can act as a complement to other types of structural and functional testing methodologies.

Visser et al. [18, 17] found that random testing, in terms of coverage, execution time, and memory used is competitive with model checking which in practice performs similarly to bounded exhaustive, and with variations with and without state matching, symbolic execution, and abstraction of states.

In addition, many refinements can be added to improve the basic randomized testing strategy. Randomized testing is the basis of the lower level of the Nighthawk tool [3]. Andrews et al. applied genetic algorithms to generate random unit test input data. This research has shown that random unit testing is an effective testing approach and Nighthawk is able to achieve high coverage of complex Java units.

Pacheco et al.'s work on the Randoop system [16] adopts a similar idea to that used in Korat [11], which performs isomorphism breaking to avoid executing essentially the same test case twice. Randoop is a test framework which automatically generates Java unit testing code using a feedback-directed random test generation approach. The fundamental algorithm used in Randoop is one which uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and invalid inputs. Feedback-directed random testing has shown promising results in quickly finding errors in widely used complex applications. The authors pointed out combining random and systematic approaches can result in techniques that retain the best of each approach. This inspires the implementation of BOBW described below, but Randoop's strategy is optimized for generating short test cases, rather than the long test cases that in [4] are shown to be more cost-effective. Another problem with Randoop's approach is it generates new test cases and then checks whether they have been executed before. As the process proceeds, more and more test cases will be discarded.

2.4 Mutation

One problem of designing testing experiments is that real programs with appropriate numbers of real faults are hard to find and hard to prepare appropriately. For example, it is hard to prepare faulty and correct versions. Even when actual programs with real faults are available, whether these faults are numerous enough to make the experimental results achieve statistical significance often becomes another problem. Many scholars have taken approaches of introducing faults into programs to produce faulty versions. We can introduce faults by hand or by automatically generating variants of the code. Generally speaking, we view an automatically-generated variant as the result of applying an operator to the code. The operators used in such a way are called *mutation operators*. The resulting faulty versions are so called *mutants* and the general technique is called *mutation* or *mutant generation*.

The idea of using mutants to measure test suite adequacy was originally proposed by DeMillo et al. [8] and Hamlet [9], and explored extensively by Offutt [15]. Andrews et al. [2] compare the fault detection ability of test suites on hand-seeded, automatically generated, and real-world faults. The experimental results have shown that mutants, when using carefully selected mutation operators and after removing equivalent mutants, can provide a good indication of the fault detection ability of a test suite [2]. Therefore, mutants can be good reflections of actual faults when assessing the behavior of testing techniques.

2.5 Andrews et al.'s Approach

In order to compare explorative strategies, Andrews et al. [4] defined canonical forms of unit tests and gave precise definitions of the search spaces and strategies. Those precise definitions provide a solid foundation to compare automated unit testing strategies.

2.5.1 Unit Test Canonical Forms

Andrews et al. show that every Java unit test case has a *canonical form*, a simplified form into which it can be transformed which is equivalent to the original. The reason for introducing canonical forms into Java unit test cases is as long as explorative strategies can generate and run all canonical form test cases, they can effectively perform any unit test case.

A *Java unit test case* is defined as a sequence of Java statements which would compile correctly when given as the body of a method. A test case T *terminates unsuccessfully* or *fails*, if it throws an uncaught exception and otherwise we say T *terminates successfully* or *succeeds*. 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.

Figure 2.1 shows an example of a Java unit test for a hypothetical Tree data structure and some equivalent canonical forms. In this thesis, we will focus on the bottom

Figure 2.1 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. [4]

```
(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];
```

canonical form (d) which is called *Canonical Form 3* [4].

In a unit test in canonical form 3, we define a *value pool* which stores all values in an array of all parameters for a method. This makes canonical form 3 particularly easy to generate automatically. Given initial decisions, each of its statements can be generated by choosing a sequence of integers. The initial decisions include how big the value pools are and what initial values to put into primitive type value pools.

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

- A first part in which an array variable which stores value pool elements is allocated. No more than one variable is declared of any given type. For example, `double[] doubleValuePool = new double[200]` declares a value pool for `double` of size 200.

- A second part in which constant values are assigned to elements of primitive type value pools; for instance, "`intValuePool[5] = 23`"

- A third part in which all statements are *array-canonical statements* [4].

- An assert statement of the form `assert` x, where x is a variable.

Here, we give formal definitions of *array-canonical statements* which act as a third part of canonical form 3. Before we define array-canonical statements, we first need to define *array-canonical method call.* An *array-canonical method call* is defined 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 [4].

An *array-canonical statement* is defined recursively as follows. S is an array-canonical statement if either:

- It is of the form $x[i] = e$ or 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.

To conclude, we can say that every Java unit test case can be converted into a canonical form which consists of a piece of code initializing value pools, and a sequence of method calls (including calling constructors) which use value pools as a source of target and parameter values, and a destination of return values. This becomes one basis for our implementation of the test program. Therefore, the only three factors that affect generating unit test cases are choosing value pool sizes, choosing initial values for primitive type value pools, and generating a sequence of integers.

2.5.2 Formal Definitions of Strategies

In order to precisely compare explorative strategies, we give the formal definitions of bounded exhaustive, randomized, and best-of-both-worlds.

2.5.2.1 Test Context

Each test strategy is relative to a test context. A *test context* consists of the following pieces of information:

- The set of methods to call, M_c.
- The set of types of interest, T_I. This should include both primitive types (including the wrapper class of primitive types in Java) and classes that are targets, parameters and return values of the methods to be called.

- For each type $t \in T_I$, the size of the value pool is defined as *vps(t)*. This is the number associated with the parameter type. The numbers from 0 to *vps(t)* - 1 can act as parameters of a method call.
- For each primitive type $t \in T_I$, the initial values of value pool elements.

2.5.2.2 Method Call Tuples

Andrews et al. introduce method call tuples to encode and abstract information about method calls. A *parameter tuple* for a method or constructor m is defined to encode the parameters to the call as a sequence of integers. The authors 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. Here, we quote parameter tuple representations for different types of method calls from [4]. In the following, V_k represents the value pool for type t_k.

- If m is a <u>static</u> method of class C with k parameters of types $t_1, \ldots, t_k$ and a <u>void</u> 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 <u>static</u> method of class C with k parameters of types $t_1, \ldots, t_k$ and a <u>non-void</u> 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
  $V_{k+1}[i_{k+1}] = C.m(V_1[i_1], \ldots, V_k[i_k])$.

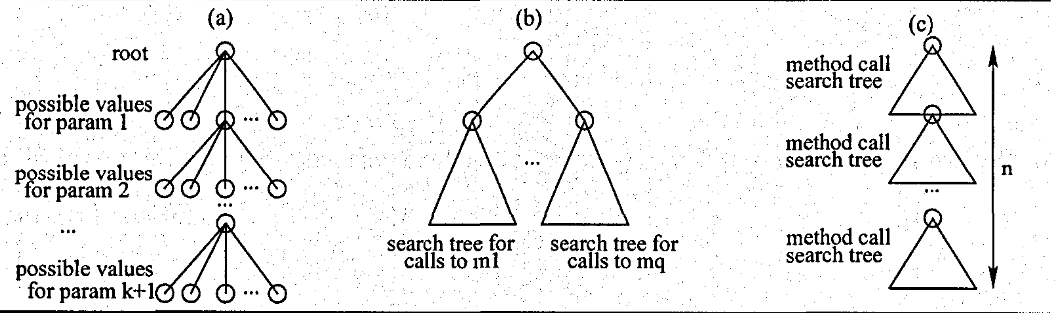
- If m is a <u>constructor</u> of class t_{k+1} with k parameters of types $t_1, \ldots, t_k$, a parameter tuple is a tuple of integers $\langle i_1, \ldots, i_k, i_{k+1} \rangle$, where each i_j is between 0 and $vps(t_j) - 1$ inclusive. The parameter tuple represents the call $V_{k+1}[i_{k+1}] = new\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

- If m is a <u>non-static</u> 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 <u>void</u> 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 <u>non-static</u> 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 <u>non-void</u> 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 $V_{k+2}[i_{k+2}] = V_{k+1}[i_{k+1}].m(V_1[i_1], \ldots, V_k[i_k])$.

What we can conclude here is that given a test context in which value pools have been defined, we can represent any parameter list by a sequence of integers: one integer representing the method and others representing the target, parameters and return value. We will treat the target and return value of a method call, if any, as "virtual parameters" in positions $j = k + 1$ and $j = k + 2$.

Figure 2.2 Search trees. (a): parameter value search tree. (b): method call search tree. (c): explorative strategy search tree.


(a)
root
possible values for param 1
possible values for param 2
...
possible values for param k+1
(b)
search tree for calls to m1
search tree for calls to mq
(c)
method call search tree
method call search tree
method call search tree
n

2.5.2.3 Search Trees

Search tree is the essential concept that Andrews et al. use to precisely define explorative strategies. Given a test context K, we can define three classes of search trees: the parameter value search tree for a given method, the method call search tree for K, and the explorative strategy search tree for K. Figure 2.2 illustrates these three classes of search trees.

A path from the root of this tree to any leaf of the parameter value search tree for m encodes one method call tuple for m. We can 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 given 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 [4].

Andrews et al. then define the *explorative strategy search tree for K for depth n* recursively as follows.

1. The tree for depth 0 is the tree with just a single root node.
2. 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.

2.5.2.4 Test Strategies

Although there is a lot of previous research on how to improve bounded exhaustive strategy, the basic idea for bounded exhaustive testing strategy is exhaustively testing all valid inputs up to a specific size or bound. Therefore, in the thesis we only consider the naive bounded exhaustive testing strategy. The *(naive) bounded exhaustive test strategy for length n*, or BE(n), is defined as the strategy that traverses the explorative strategy search tree in a depth-first manner, executing the corresponding test case whenever it reaches a leaf.

The *randomized test strategy for length n and repetitions q*, or R(n, q), is defined as the strategy that, q times, randomly selects a path from root to leaf of the explorative strategy search tree, and executes the corresponding test case.

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.

In [4], Andrews et al. analyzed the uniform and non-uniform distributions of failure as well. According to their analysis, it is likely that failure does not distribute uniformly. This means in most cases, the failing test cases will not be spread evenly in the explorative strategy search tree. The reason is that a fault in a method will lead to a failure only if the method is executed, or only if it is executed after certain patterns of method calls. Therefore, the nodes in the search tree corresponding to failing test cases are likely to cluster in certain areas of the tree. From this analysis, Andrews et al. state that due to the risk of clustering of failing test cases, R is likely to outperform BE except at low failure densities.

Chapter 3

Test Program

In this chapter, we go into detail about the design and implementation of the test program. We will describe the architecture, some important design decisions, and important classes and interfaces of the test program. Additionally, since the test program implements BE, R and BOBW, we will illustrate the algorithms for implementing these three test strategies.

3.1 Background and Motivation

As mentioned in the Related Work chapter, Andrews et al. (See section 2.5) provide solid theoretical foundations for comparing automated unit testing strategies. The concepts and algorithms of their work basically motivate our test program. First and foremost, the test program implements BE, R and BOBW. We will describe the algorithms for the respective implementations. Second, the test program implements test context, which provides all necessary information to run test cases. Third, the test program abstracts information for method calls and constructors in order to treat them homogeneously. Fourth, the test program is able to run test cases for different test strategies and depths. Last but not least, we need specific test oracles to verify whether the test case succeeds or fails. In addition to these functional requirements, the test program should be flexible, easy to maintain, and adaptive to change. This is mainly reflected in the design of our test program.

3.2 Introduction

The test program we developed is called *Universal Test* or *UT*. UT is a Java program developed in Java Development Kit (JDK) 1.5.0. It is tested and compatible with all versions of JDK 1.5 and 1.6. UT is able to run on any Operating System that JDK supports. As the Java programming language organizes its source code into packages, UT has 4 packages with total of 17 source files.

Besides the Java program, the other part of UT is shell scripts which drive the Java program with different input arguments and subject units. We use shell scripts to record the CPU time of the Java program execution as well. The set of shell scripts is written in Bash and contains 6 script files. The shell scripts should be able to run on any UNIX-like operating systems.

To start running UT, you need to specify three program arguments: strategy name, depth (the length of test case), and the total number of test cases to run. These three arguments are taken into the script `OORunUniversalTest.sh`. For example, if you want to run randomized testing strategy with depth = 5 and total 1000 test cases, you can type:

```
OORunUniversalTest.sh -r 5 1000
```

"`-r`" stands for randomized testing strategy. Other options are "`-be`" which stands for bounded exhaustive and "`-bobw`" which stands for best-of-both-worlds.

3.3 Architecture

We use a Unified Modeling Language (UML) package diagram to illustrate package organization in Figure 3.1. As shown in the figure, there are 4 packages all starting with `cs.uwo`. Package `cs.uwo` contains the main method in the Main class which is the entrance to the test program. The main method takes in the program arguments, and then sets up and initializes "test context" (called `TestInfo` in UT) objects. When the initialization is finished, it invokes the corresponding test strategy with the specified depth and number of test cases to run, according to the program arguments. The package `cs.uwo.util` is a utility package consisting of `ClassFinder`, `Debugger`, `IOHandler` and `LogAnalyzer` classes. Those helper classes facilitate us debugging the program, writing output files and analyzing log files. The most important two packages are `cs.uwo.testenvironment` and `cs.uwo.strategy`. Briefly speaking, the package `cs.uwo.testenvironment` is the implementation of the test context and method tuples in Andrews et al.'s work. It provides all necessary information to run test cases. In the package `cs.uwo.strategy` are implementations of the three test strategies that this thesis focuses on, bounded exhaustive, randomized and best-of-both-worlds. The test strategies invoke methods in `cs.uwo.testenvironment` and verify the output.

3.4 Design

A good design plays a critical role in software development. A good design makes the program flexible and easy to maintain. Since the system requirements constantly change, we sometimes need to refactor the code in order to better adapt to those changes. During the design phase, we need to consider potential requirements in our design decisions. In UT, we carefully design methods, classes, interfaces and packages to make it extendable for future research requirements. Two important packages in UT are `cs.uwo.testenvironment` and `cs.uwo.strategy`. In this section, we talk about the design of these two packages and their classes.

3.4.1 Package `cs.uwo.testenvironment`

The package `cs.uwo.testenvironment` consists of 2 interfaces: `TestInfo` and `ThingToCall`, and 4 classes: `CallDescription`, `TestCase`, `TestInfoImp`, and `ThingToCallImp`. Figure 3.2 shows the classes and interfaces, and their relationships in package cs.uwo.testenvironment using a UML class diagram. In Figure 3.2, we only show important fields and methods of a class or interface. Accessor and auxiliary methods are not shown in the figure.

TestInfo is the implementation of "test context" (see 2.5.2.1). Java has 8 primitive data types: byte, short, int, long, float, double, char, boolean. For each primitive data type, TestInfo has a corresponding value pool. Note here we treat String as a primitive data type as well. Therefore, there are total of 9 primitive type value pools in the TestInfo class. We use a vector to represent each primitive type value pool. TestInfo also provides an add method to add values to the corresponding primitive type value pool. In Figure 3.2, it only shows the _intValuePool vector, which is the value pool for primitive type int, and the addIntValue() method which is used to add int values to the int value pool. In addition to primitive type value pools, the TestInfo class uses a HashMap as a class value pool. The class value pool stores values for classes which are not primitive data types. Here we should note that the Java compiler automatically wraps the primitive to an object, if we use a primitive where an object is expected. The Java platform provides *wrapper* classes for each of the primitive data types. Therefore, we put values of the primitive type wrapper classes into corresponding primitive type value pools as well. Another important part of the TestInfo class is to provide necessary information to make method calls. The TestInfo class uses a vector to store all methods (each method is wrapped in a ThingToCall class which we will talk about later in the subsection) for a given class. With a ThingToCall index and a vector of parameters including virtual parameters (see section 2.5.2.2), the method callThingNumber is used to locate the corresponding ThingToCall and pass virtual parameters to make the method call.

The `ThingToCall` interface is an interface abstracting a method or constructor. The Java reflection mechanism provides us all necessary information to abstract a method or a constructor. For a method, we can get an array of parameter types, the return type, and the declaring class of the method, which is considered as the receiver class. For a constructor, we can get the same information except the return type because a constructor doesn't have a return value. The `getNumParameters` method in the `ThingToCall` interface gets the (virtual) number of parameters of the thing to call. The (virtual) number of parameters should be calculated as follows:

1. Let n = number of declared parameters of the method or constructor.
2. If the thing to call is a non-static method, then n = n+1.
3. If the thing to call is a constructor or has a non-void return value, then n = n+1

The `makeCall` method is used to make a call to the thing to call. It has two parameters. The first one is a `TestInfo` object which is used to locate the `ThingToCall`, get values from the appropriate value pool, and make actual method call. The other parameter is named `valueIndices`, an integer vector of (virtual) parameters. The number in `valueIndices` indicates the index of the value from the value pool that is to be used as the virtual parameter. The vector of integers should consist of `getNumParameters()` integers, each one in the correct range. Let us assume that the thing to call has n declared parameters. The `makeCall` algorithm is described as follows:

- If the thing to call is a non-static method, then choose as the receiver the k-th value from the appropriate value pool, where k is the n-th element of valueIndices.

- For parameter i, where i is between 0 and n-1, choose as the parameter the k-th value from the appropriate value pool, where k is the i-th element of valueIndices.

- Call the method or constructor using Java reflection.

- If the call threw a Throwable, then return that Throwable; otherwise continue.

- If the thing to call is a method with a void return value, then return null; otherwise continue.

- If the thing to call is a non-static method with a non-void return value, then place the return value in element k of the appropriate value pool, where k is the n+1-th element of valueIndices.

- If the thing to call is a static method with a non-void return value, then place the return value in element k of the appropriate value pool, where k is the n-th element of valueIndices.

- If the thing to call is a constructor, then place the new object in element k of the appropriate value pool, where k is the n-th element of valueIndices.

- Return null.

The makeCall method returns any throwable if there is any; otherwise null.

CallDescription is a class representing a method call. It wraps the corresponding ThingToCall index and parameter indices. Given a TestInfo object, we can call the actual method or constructor wrapped in the `CallDescription` object. The `TestCase` class contains a vector of call descriptions which are added when building the test case. Given a `TestInfo` object, we can run the test case using the "execute" method. If there are any `Throwables` thrown out when executing the call description, the `TestInfo` will store them. These `Throwables` will be used later to compare one test case execution to another.

3.4.2 Package `cs.uwo.strategy`

The package `cs.uwo.strategy` contains implementations of BE, R and BOBW. Figure 3.3 depicts fields, methods, classes, interfaces and their relationships using a UML class diagram. Here we list all classes and interfaces with only important methods and fields of each class. The Strategy class is an abstract class which provides fields and methods required by all sub-classes. It has three very useful methods: `setup`, `compare` and `executeTestCase`. Given a `TestInfo` object and a `Class` object of the subject unit, the setup method extracts all public methods and constructors, add them to the `TestInfo` object as `ThingToCall` objects, and initializes value pools. When the test case is built, we can call the `executeTestCase` method to run the test case.

As mentioned before, a test oracle should be used to test whether the test case fails or not. The `compare` method in the `Strategy` class is used as a test oracle. We use two criteria to build our test oracle. The first one is to compare primitive type value pools between the "gold" version which is the original subject unit, and the "faulty" version which is the mutated version. If any value of these two value pools is not equal, we can assert that this is a failing test case; otherwise, this is a passing test case. The `primitiveValuePoolsEqual` method in the `TestInfo` class compares primitive type value pools and returns `true` if every element in every value pool for every primitive type is equal to the corresponding element in other; `false` otherwise. The second criterion is to compare the number of throwables between the "gold" version and "faulty" version. The `TestInfo` class stores a vector of all throwables thrown during the test case execution. We compare the sizes of the two vectors of two `TestInfo` objects. If they are not equal, we can assert that this is a failing test case; otherwise a passing test case.

The reason that we are implementing the strategies to take a "gold" and "faulty" version is because we are doing experiments to measure the effectiveness of the testing. Therefore, this requires us to implement the strategies differently. However, for general purposes, what a developer would use is some implementation that just takes a single version.

The `BobwStrategy`, `BoundedExhausiveStrategy`, `IterationRandomStrategy`, and `RecursionRandomStrategy` classes all inherit from the `Strategy` class. They implement the BOBW, BE and Random (iteration and recursion) strategies respectively. The implementations will be explained in detail in the following sections.

3.5 Implementation of Bounded Exhaustive Test Strategy

The BoundedExhausiveStrategy class implements the bounded exhaustive strategy. Briefly speaking, the algorithm used to implement BE is a mutual recursion of two methods, *createRunAllTestCases* and *completeTestCase*. The mutual recursion means that createRunAllTestCases is a recursion itself and it calls completeTestCase, and completeTestCase is a recursion itself and it calls createRunAllTestCases. Given a certain depth and an input test case, the createRunAllTestCases method generates and runs all test cases that extend the input test case. The completeTestCase method is similar, but it also takes a partially completed call description as input. It creates and runs all test cases that are extensions of the input test case plus the call description so far.

At the beginning of the BE strategy, it creates an empty test case and passes it to the method createRunAllTestCases. The createRunAllTestCases method takes a depth and a TestCase object containing 0 or more completed CallDescriptions as inputs. It creates and runs all test cases that are extensions of the input test case, up to the depth bound. For example, if the depth is 3 and the input test case contains call descriptions A and B, createRunAllTestCases will add all possible call descriptions to the end of the existing but not completed test case, and run all of them.

In the createRunAllTestCases method, it first judges whether the depth of the test case (i.e. length of method calls) so far is equal to the depth we want to build in the test case. (We should note here methods or constructors have been wrapped in `ThingToCall` objects of the `TestInfo` object.) If it's not equal, the number of call descriptions added (i.e. test case length) so far is less than the number of call descriptions we expected. We need to create and add new call descriptions to the end of the test case. For all the `ThingsToCall`, we create new call descriptions and call the `completeTestCase` method to add a `CallDescription` object in the depth so far. If the depth so far is equal to the depth we wanted, we will run the test case on both the "gold" `TestInfo` object and "faulty" one. Then we compare those two `TestInfo` objects. If the comparison returns false, we can assert this is a failing test case. The BE strategy then writes the failing test case information to a log and returns. Otherwise it continues.

The second method is `completeTestCase`. It is used to add arguments to call descriptions until the call descriptions have sufficient arguments. Once the arguments of a call description are completely added, the call description will be added to the input test case. If the arguments we add to the call description so far equal to the arguments wanted (including the virtual parameters), the call description will be added to the test case. Therefore, the test case depth so far will be incremented and the `createRunAllTestCases` method will be invoked to try to create and run test cases. We should note the entry of the `completeTestCase` method is the `createRunAllTestCases` method, and here the `completeTestCase` method jumps back to the `createRunAllTestCases` method with depth increased. Once the `createRunAllTestCases` method returns, we will remove the last call description. If the arguments so far are not equal to the arguments that the call description should have, the algorithm will add more arguments to the call description. It first gets the `ThingToCall` index for the call description. Then, given the argument index so far and the ThingToCall index, the algorithm can retrieve the value pool size from the TestInfo object. The bounded exhaustive algorithm needs to add every value index, up to the value pool size, to the call description. Then, an argument is added to the call description and we recursively call the completeTestCase method to add more arguments. Finally, after the recursion returns, the algorithm removes the last argument.

To sum up, the BE implementation iterates all value indices in the corresponding value pool as the arguments, and explores every public method or constructor (wrapped in ThingToCall objects and CallDescription objects) of the given class.

3.6 Implementation of Randomized Test Strategy

There are two algorithms for implementing the randomized test strategy: a recursion algorithm and an iteration algorithm. In the randomized test strategy, we need to specify the number of test cases to run. The strategy uses the Random class in JDK to generate a random number. Our implementation of the randomized test strategy uses the iteration algorithm instead of the recursion algorithm.

The iteration algorithm uses a different approach rather than the recursion. Figure 3.4 shows the iteration random algorithm. It uses three loops. The outer loop is used to control the total number of test cases. The middle loop is used to control the depth of the test case. The inner loop is used to control the arguments that are expected in the call description. The algorithm is very straightforward, as depicted in Figure 3.4. Like the recursion random algorithm, the iteration algorithm first randomly selects the `ThingToCall` index and then randomly selects the argument index within the value pool size. After arguments and call descriptions are added to the test case, it executes the test case.

3.7 Implementation of Best-of-Both-Worlds Test Strategy

To implement BOBW, taking the index of each test case, it is necessary to generate all numbers from 0 to $z-1$ (see section 2.5.2.4) without repetitions. To achieve this, we 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. The test case indices are so large that they cannot be represented by Java primitive type numeric variables. The Java standard `BigInteger` class is used to represent the test case indices. The representation takes a number of bits proportional to $\log(z)$, which is $n\log(j)$.

The `BobwGenerator` class is used to generate the BOBW test cases. The large prime number that we pick is a large Mersenne prime number found by Lucas in 1876. The `BobwGenerator` class has two public methods: `hasMoreTestCases` and `getNextTestCase`. `hasMoreTestCases` judges whether the current test case index is equal to zero and returns true; otherwise it returns false. The getTestCase method returns a test case generated by the BOBW algorithm mentioned above. Let n be the number of method calls. The process of extracting the actual test case from its index takes n steps of length proportional to $\log(j)$. Note here that the process of generating and running a test case for BE and R also takes time proportional to n.

The `BobwStrategy` class simply generates a new `BobwGenerator` object and repeatedly invokes the `getNextTestCase` method to generate the BOBW test case. After the test case is generated, it runs the test case.

Figure 3.1 Package organization of Universal Test

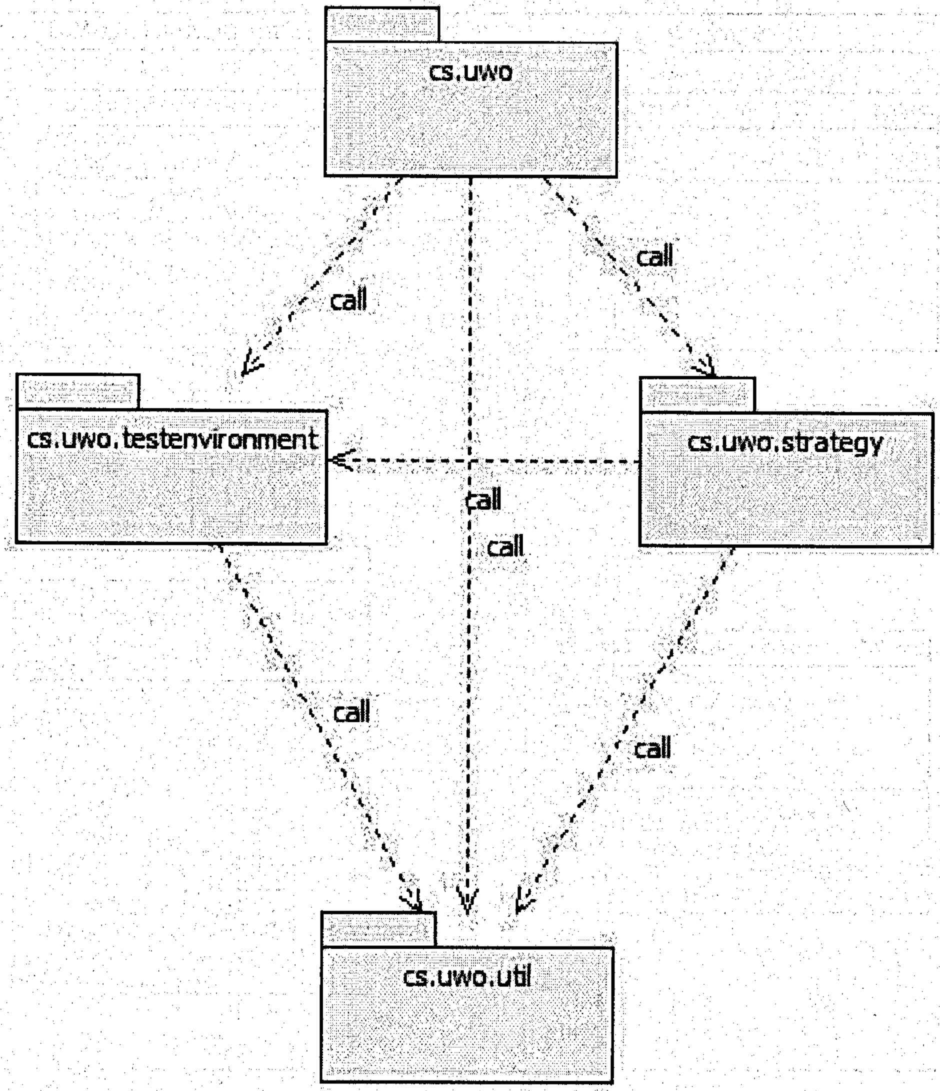


Figure 3.2 Class diagram of `cs.uwo.testenvironment` package

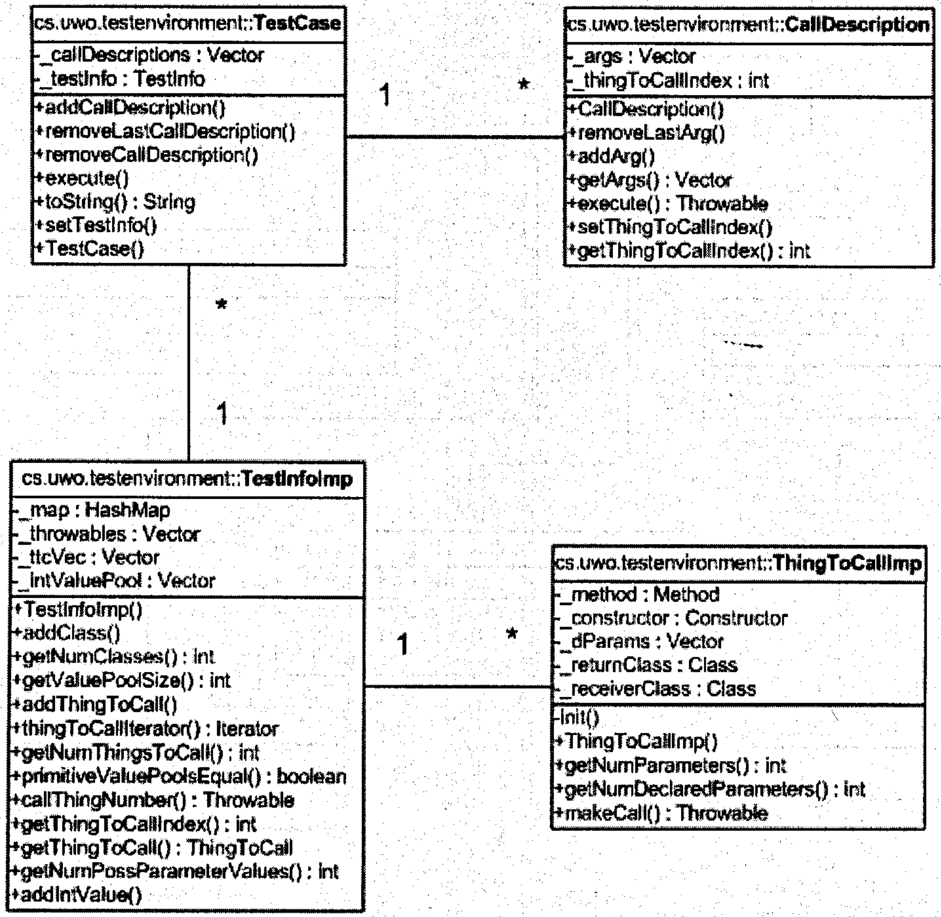

cs.uwo.testenvironment::**TestCase**
-_callDescriptions : Vector
-_testInfo : TestInfo
+addCallDescription()
+removeLastCallDescription()
+removeCallDescription()
+execute()
+toString() : String
+setTestInfo()
+TestCase()
cs.uwo.testenvironment::**CallDescription**
-_args : Vector
-_thingToCallIndex : int
+CallDescription()
+removeLastArg()
+addArg()
+getArgs() : Vector
+execute() : Throwable
+setThingToCallIndex()
+getThingToCallIndex() : int
1
*
*
1
cs.uwo.testenvironment::**TestInfoImp**
-_map : HashMap
-_throwables : Vector
-_ttcVec : Vector
-_intValuePool : Vector
+TestInfoImp()
+addClass()
+getNumClasses() : int
+getValuePoolSize() : int
+addThingToCall()
+thingToCallIterator() : Iterator
+getNumThingsToCall() : int
+primitiveValuePoolsEqual() : boolean
+callThingNumber() : Throwable
+getThingToCallIndex() : int
+getThingToCall() : ThingToCall
+getNumPossParameterValues() : int
+addIntValue()
1
*
cs.uwo.testenvironment::**ThingToCallImp**
-_method : Method
-_constructor : Constructor
-_dParams : Vector
-_returnClass : Class
-_receiverClass : Class
-init()
+ThingToCallImp()
+getNumParameters() : int
+getNumDeclaredParameters() : int
+makeCall() : Throwable

Figure 3.3 Class diagram of `cs.uwo.strategy` package

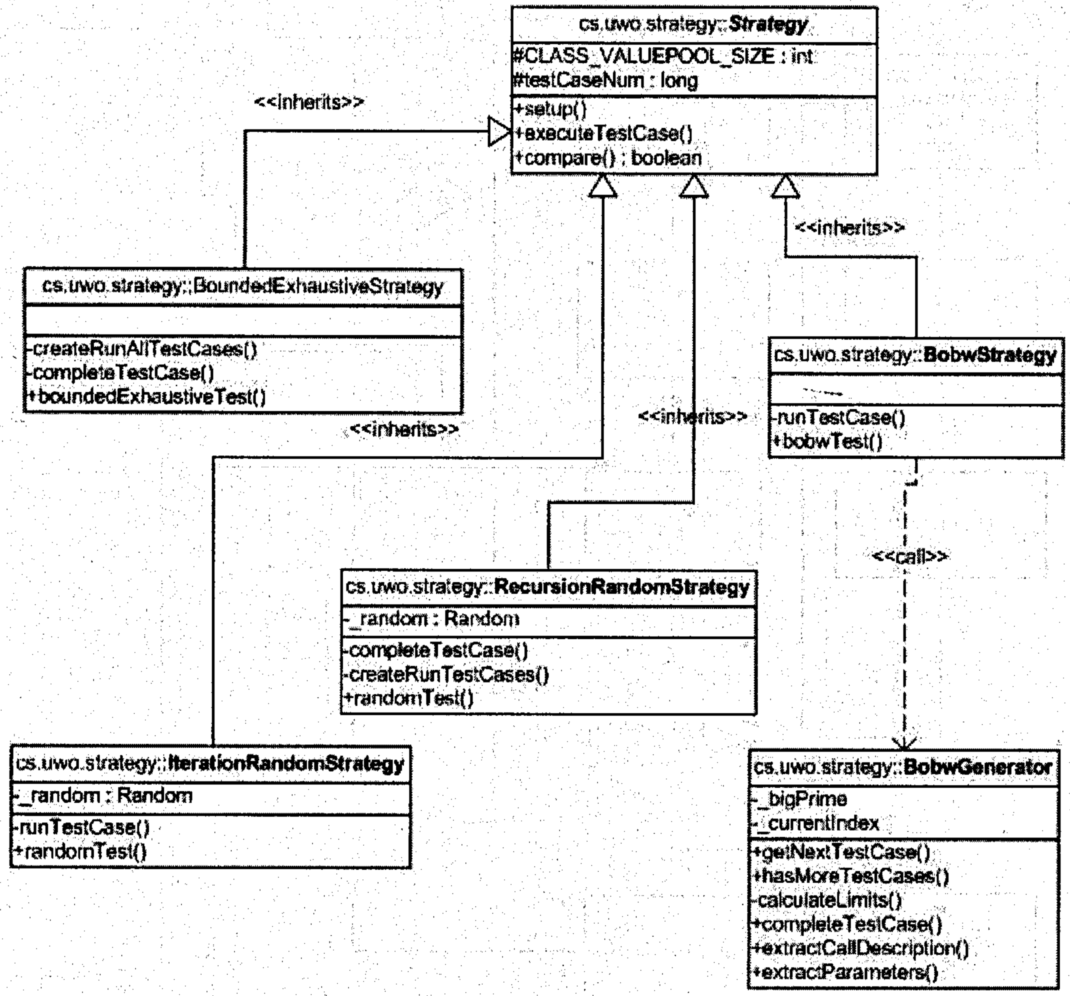
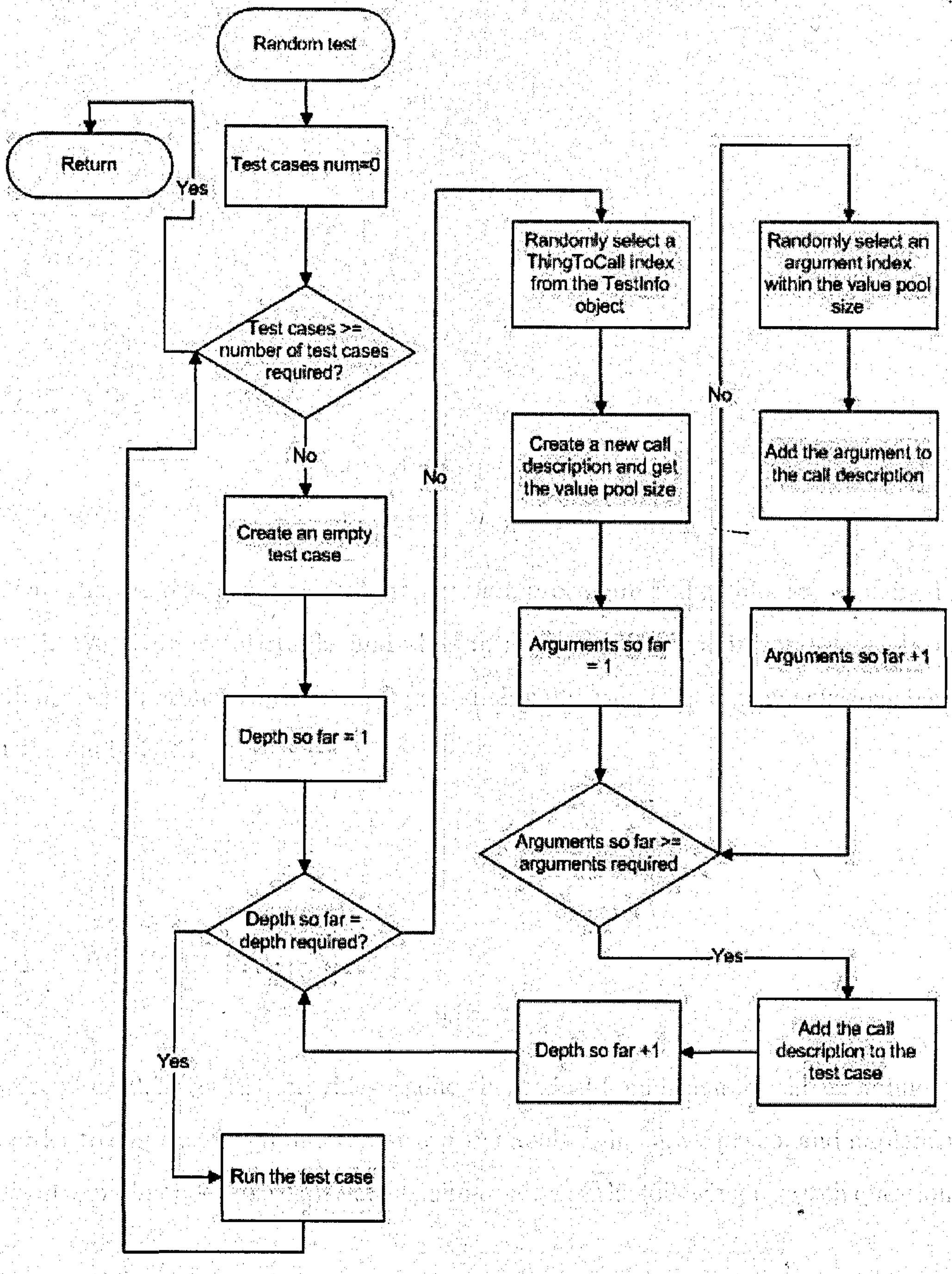


Figure 3.4 Iteration random algorithm of the random test strategy


Random test
Return
Test cases num=0
Yes
Test cases >= number of test cases required?
No
Create an empty test case
Depth so far = 1
Depth so far = depth required?
Yes
Run the test case
No
Randomly select a ThingToCall index from the TestInfo object
Create a new call description and get the value pool size
Arguments so far = 1
Arguments so far >= arguments required
No
Randomly select an argument index within the value pool size
Add the argument to the call description
Arguments so far +1
Yes
Add the call description to the test case
Depth so far +1

Chapter 4

Experiments

In this chapter, the experimental preparation, procedure and results are explained in detail. We give several graphs and plots to illustrate the experimental data. In addition, the experimental data are analyzed. Finally, we draw some conclusions based on the analysis.

4.1 Motivations

Andrews et al.'s work [4] provides a solid theoretical foundation for the experiments. In order to empirically ground the theory and analysis in [4], we design and implement several experiments. The experiments aim to answer the following research questions:

- How many mutants does R detect compared to BE? In order to compare the abilities to detect failures, we need to figure out the number of mutants killed by R and by BE.

- Does R detect any mutants that BE does not detect and vice versa? If there is some mutant that R detects but BE does not, we need to find out what that mutant is and give an explanation of why R is able to detect that specific mutant but BE cannot, and vice versa.

- How long does it take for R (BE, BOBW) to find its first failure, all in terms of number of test cases and in terms of CPU or clock time? In order to compare the effectiveness of finding failures for R, BE and BOBW, we need to record both the number of the first failing test case and the CPU time or clock time. If one of these strategies takes least time or least number of test cases to find the first failure, we may say this is the most effective strategy in finding the first failure.

- How does the length of the test cases affect the length of time to first failure and the number of test cases to first failure? In [4], it shows that not only do longer test cases reveal higher failure density, but often reveal more cost-effective testing. We want to provide an empirical study to see how the length of the test cases effects the length of time to first failure and the number of test cases to first failure.

- Are the above numbers consistent with the theoretical analysis in [4]? The experimental data would reveal the consistency or inconsistency with the theoretical analysis in [4]. If there is any inconsistency, we will give reasonable explanations of it.

Figure 4.1 Data concerning experimental subjects.

Unit	SLOC	Mutants Compiled	Mutants Non-Equiv.
ArrayList	150	100	47
EnumMap	239	100	0
HashMap	360	100	31
HashSet	46	41	6
Hashtable	355	100	46
IHashMap	392	100	100
LHashMap	103	74	6
LHashSet	9	0	0
LinkedList	227	100	46
PQueue	203	100	41
Properties	249	100	1
Stack	17	33	28
TreeMap	562	100	25
TreeSet	62	45	8
Vector	200	100	93
WHashMap	338	100	38
Total	3512	1293	516

Those research questions are motivations for doing the experiments. Experiments and experimental data will be explained in detail in the following sections.

4.2 Subject Units

The subject unit that I will do experiments on is a set of heavily-used Java data structure units. It is 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). Figure 4.1 shows the data concerning the experimental subjects.

4.3 Experimental Preparation

The mutants generated act as faulty versions while the original source files act as gold versions. The mutants are generated using the same mutant generator as in [2], which generates the mutants based on four types of changes: “replace operator”, “replace constant”, “negate decision” and “delete statement”.

Since the `java.util` classes often take generic type parameters, in order to simplify the experimental infrastructure, we generate a “wrapper” class for each of the `java.util` classes, which instantiates the generic type parameters to `Integer`. Each wrapper class contains 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`.

Each of the test strategies in the test program takes two `TestInfo` objects. One of the `TestInfo` objects refers to the original, “gold” implementation of the class and its methods. The other refers to a mutant implementation, the “faulty” version. The `TestInfo` object used in the experiments is one in which each primitive type value pool has two elements and each class value pool has one element. Each primitive type value pool is intialized with two distinct constants (e.g., 0 and 100 for the `Integer` value pool). The selection of the value pool size and value pool elements reflects the design of our experiments. We want non-trivial instances of `TestInfo`. However, we still want the `TestInfo` instance to be small enough that it runs efficiently and BE can reach large depths in a measurable amount of time. If we add many values (e.g. 10 values) into the primitive type value pool, this will greatly expand the width of the corresponding parameter value search tree (see Section 2.5.2.3). Therefore, BE will take a fairly large amount of time to execute test cases up to the depth that we want to compare, and it makes it infeasible to measure the effectiveness of the BE strategy.

As stated in section 3.4, each strategy generates and runs test cases on both the gold and the faulty version. Any exceptions thrown as a result of the method calls are 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 are compared directly. If the size of the exception list is different or any value in the value pools is different, this indicates that we have found a test case for which the mutant behaves differently from the "gold" version. Therefore, we assert that a failure has been found in the mutant unit. This is also referred to as "killing" the mutant.

4.4 Experimental Procedure

The experiments proceed in two phases. The first phase is to identify which mutants are equivalent and which mutants are non-equivalent. A mutant is *non-equivalent* if there are any failures on any test cases. If all test cases succeed, then the mutant is *equivalent.* This is an approximation, because it is possible that a mutant will behave differently on some test case that we have not yet run. Therefore, mutant equivalence is undecidable, and some approximation like this is needed. According to Andrews et al.[4], *failure density* is defined as a ratio between the number of failing test cases and the total number of test cases. In the second phase, we measure failure densities and compare the strategies on the non-equivalent mutants. For all the experimental procedures, we have a set of shell scripts to automate the test program and collect the experimental data.

An experiment is denoted as "strategy name(number of method calls per test case, total number of test cases)". For example, a randomized testing experiment with 10 method calls per test case and total 1,000 test cases is denoted as R(10,1000). For identifying which mutants are equivalent, we first run experiment R(10, 1000), then R(100, 1000), and then R(1000,1000). The reason for running the R test strategy first is that we believe R would be the best way to quickly identify failing test cases. In order not to bias the experiments in favor of R, if R cannot detect any failing test cases, for each such mutant, we also run BE testing with 3, 4, and 5 method calls per test case, until either a failure is detected or 30 minutes of clock time has passed.

One problem here is how we know the reason that a test case is taking too much time (more than 30 minutes) to finish. If a test case is running too long, it is either because of an infinite loop in the test case or the complexity of method calls in the test case. To solve this problem, UT (see section 3.2) writes log files with test case number, beginning, end, and timestamp of a test case so that we can easily identify whether there is an infinite loop in the test case. If there is a test case that only has the beginning statement and no end statement, and the timestamp shows the test case begins a long time (10 minutes) ago, we can assert an infinite loop occurs in the test case. If an infinite loop occurs, we terminate the process immediately in order to move on to the next mutant.

As shown in the 4th column in Figure 4.1, there are total of 516 non-equivalent mutants. 82 of the 516 non-equivalent mutants failed by going into infinite loops, rendering them infeasible for further experiments. Therefore the rest of the experiments were performed on the 434 non-equivalent mutants that did not go into infinite loops. 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 record how long R takes to finish 1000 test cases at depth n (in CPU time), how long R takes to find its first failure (in clock time and number of test cases), and how many of the test cases fail in total. We use $E(n)$ to denote the index of the earliest failure at length n.

Running BE for a complete run with the same lengths of method calls as R is infeasible, even for short lengths. Hence, we only run BE(n) for $n = 1$ to 8, stopping as soon as a failure is found or $E(n)$ test cases are run. The information collected is whether a failure was found by BE, how many test cases were run, and how much total CPU time was needed.

We also ran BOBW(n, 1000), using a similar experimental procedure as 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 record how long BOBW takes to finish 1000 test cases at depth n (in CPU time), how long BOBW takes to find its first failure (in clock time and number of test cases), and how many of the test cases fail in total.

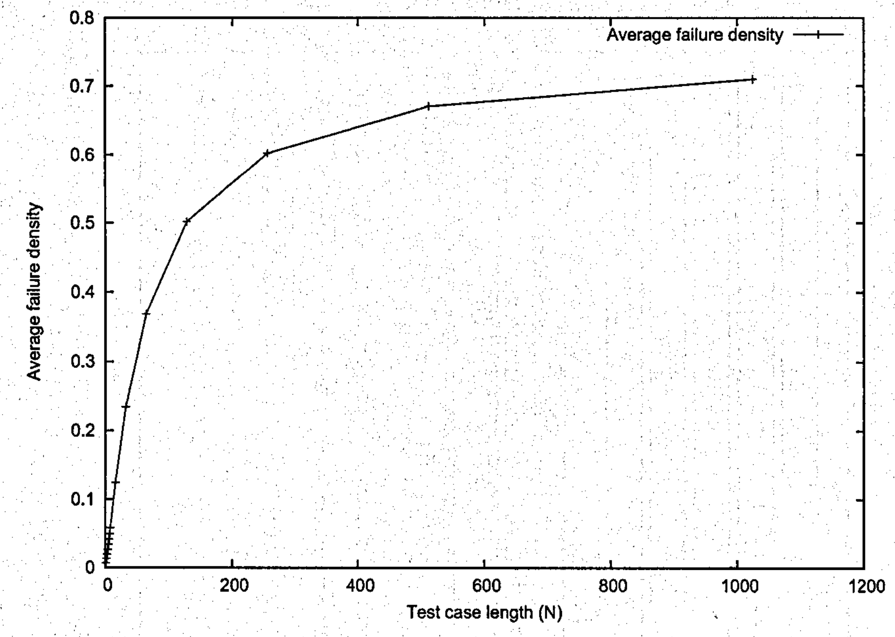
4.5 Experimental Results

In the first phase of the experiment (identifying equivalent and non-equivalent mutants), 435 mutants over all `java.util` classes are non-equivalent. This means that either R or BE is able to find a failing test case for 435 of the mutants. 434 of them are found by runs of R; only one (a mutant of `Hashtable`) is found by BE but not by R. This mutant is one which changes 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 collected can be used to measure the failure density. Figure 4.2 illustrates the failure density for the `java.util` units, averaged over all non-equivalent mutants of all mutants, as computed from the data from the runs of R. Consistent with the analysis in [4], the failure density climbs as n increases.

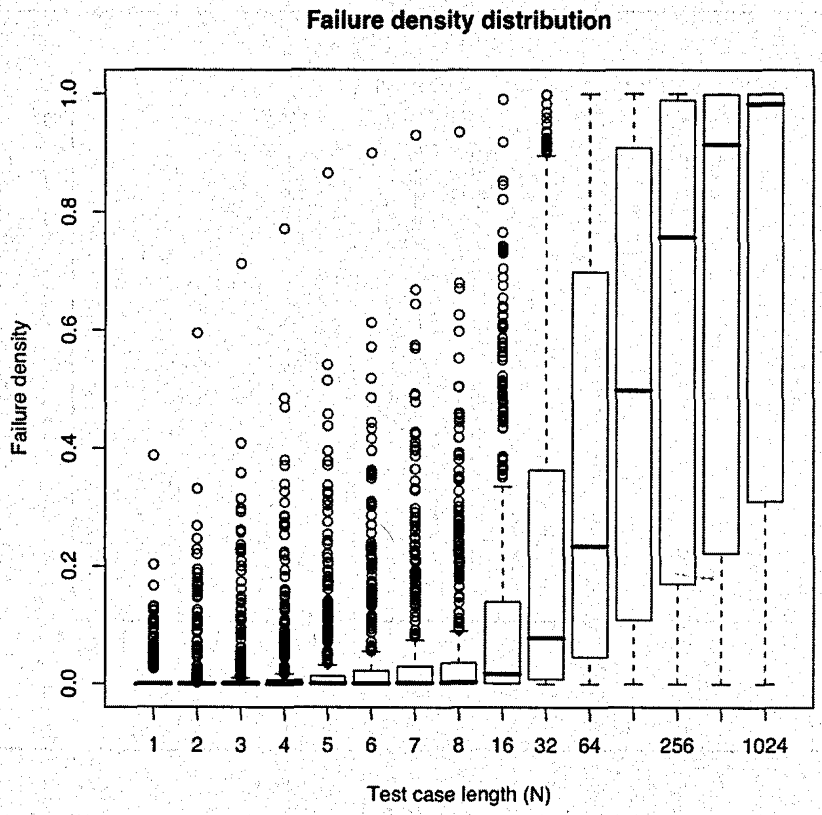
Figure 4.3 is a box plot showing the failure density for the `java.util` units, as computed with the same criteria as figure 4.2. The average failure density climbs with the increasing test case length n. It approaches 1.0 as n increases. This means as test case length increases, it becomes more and more likely that a given test case will cause a non-equivalent mutant to fail.

Figure 4.2 Failure densities for `java.util` mutants, by test case length.[4]



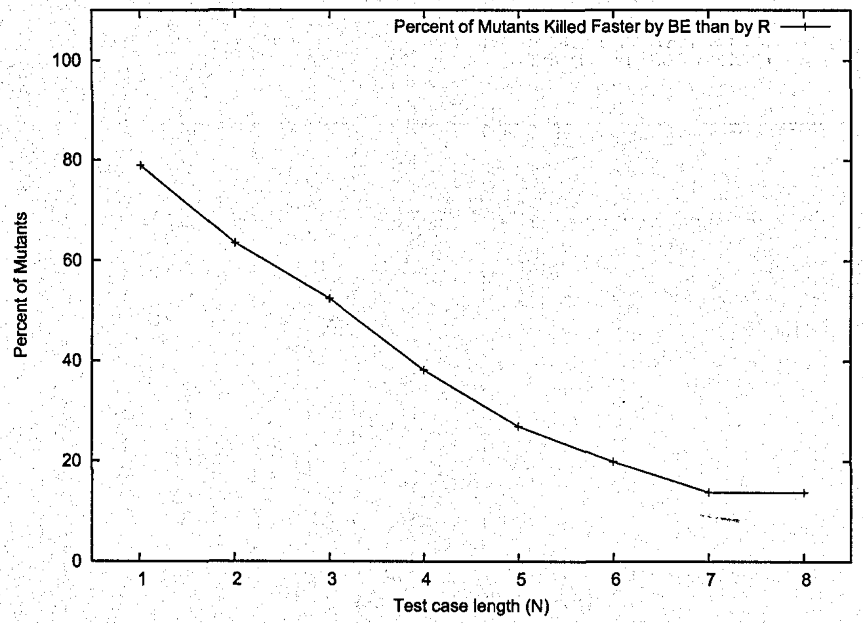
Andrews et al. theoretically analyzed the clustering of failing test cases (see Section 2.5.2.4). In order to examine whether the clustering occurs in practice, the experiment examines the situations when R(n, 1000) could kill a mutant (i.e., find a failing test case for the mutant) and BE(n) could kill the mutant in fewer test cases. If failures are evenly distributed throughout the search space, or clustered in the low level of the search space that favours BE, we would expect that BE would kill 50% or more of the mutants more quickly (in fewer test cases) than R. BE has the natural advantage of not repeating test cases, which should give it the edge when failure densities are low. Figure 4.4 depicts the comparison discussed above. BE kills over 50% of the mutants in fewer test cases than R only when the lengths of test cases are short (n = 1, 2,

Figure 4.3 Box plot for failure densities for `java.util` mutants, by test case length.



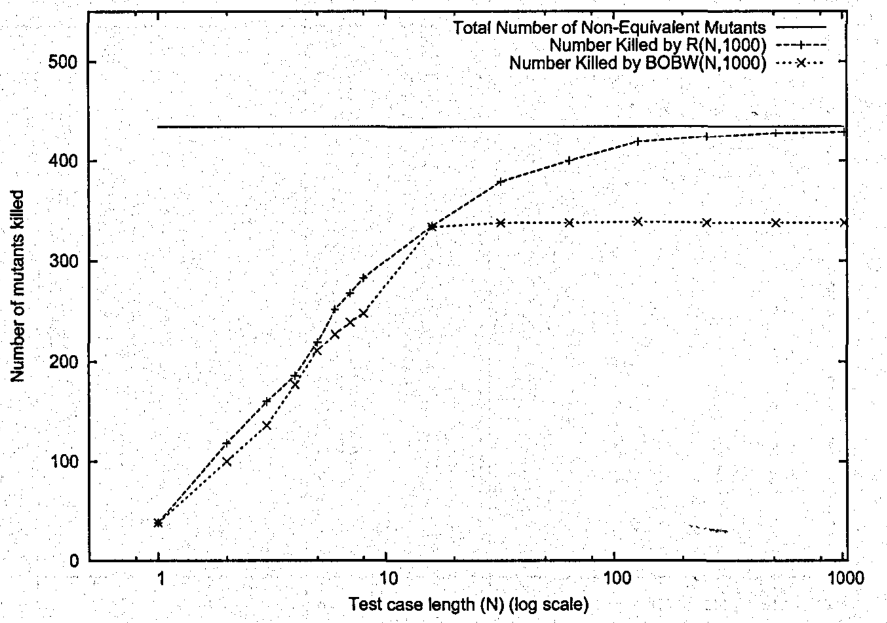
and 3), when lower failure densities are expected. At $n = 3$, BE kills close to 50% of the mutants in fewer test cases. From $n = 4$ to higher lengths, R is more effective than BE because of the combination of higher failure densities and the clustering of failures. Since we adopt different appraoches to measure the effectiveness of BE, we should note here that figure 4.4 does not indicate that BE kills fewer mutants than R. Therefore, it does not contradict the fact that a full run of BE for a given test case length, although it is often infeasible in practice, will find failures that R will not find when running the same number of test cases.

Figure 4.4 Percentage of test 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. [4]



In addition to the comparison of BE and R, the experiments also compare the number of mutants killed by R and our implementation of BOBW. Figure 4.5 shows the comparison of R and BOBW in terms of the number of mutants killed by each strategy. The solid line is the number of mutants overall, i.e. the maximum number of mutants that could be killed. Starting from test case length 1, the number of mutants killed by R consistently climbs until the test case length reaches 1024. However, it is surprising that the number of mutants killed by BOBW cannot beat the number killed by R at all test case lengths. The number of mutants killed by BOBW almost remains unchanged since the test case length 16. A possible explanation for this is that if the prime number used in the BOBW strategy is not large enough, the BOBW strategy will still select test cases that are close to each other relative to the size of

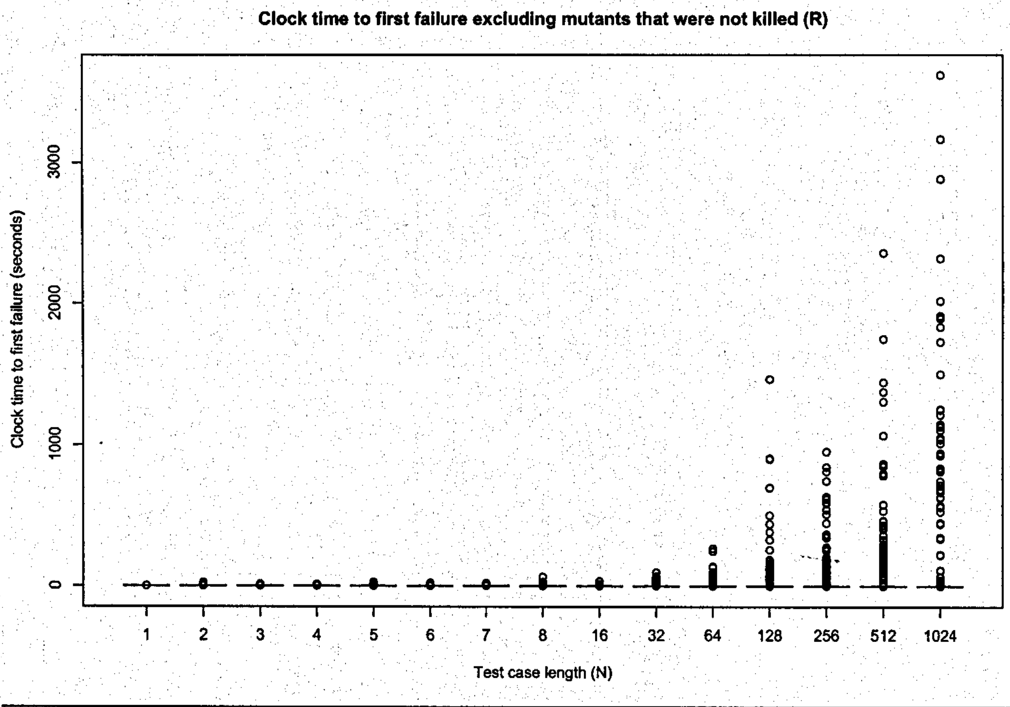
Figure 4.5 Comparison of R and BOBW in terms of the number of mutants killed (log scale x axis)



the search space. The BOBW strategy tends to randomly select test cases within the search space without replacement. If p in our implementation of BOBW is small relative to the search space, our implementation of BOBW will pick up test cases in only a small portion of the search space.

According to the experimental data and figure 4.5, the choice of constants in the implementation of BOBW has not achieved the desired properties of a pseudo-random number generator. The linear congruential random number generator might be able to be used to get the desired properties. Other ways of BOBW implementation are considered as our future work.

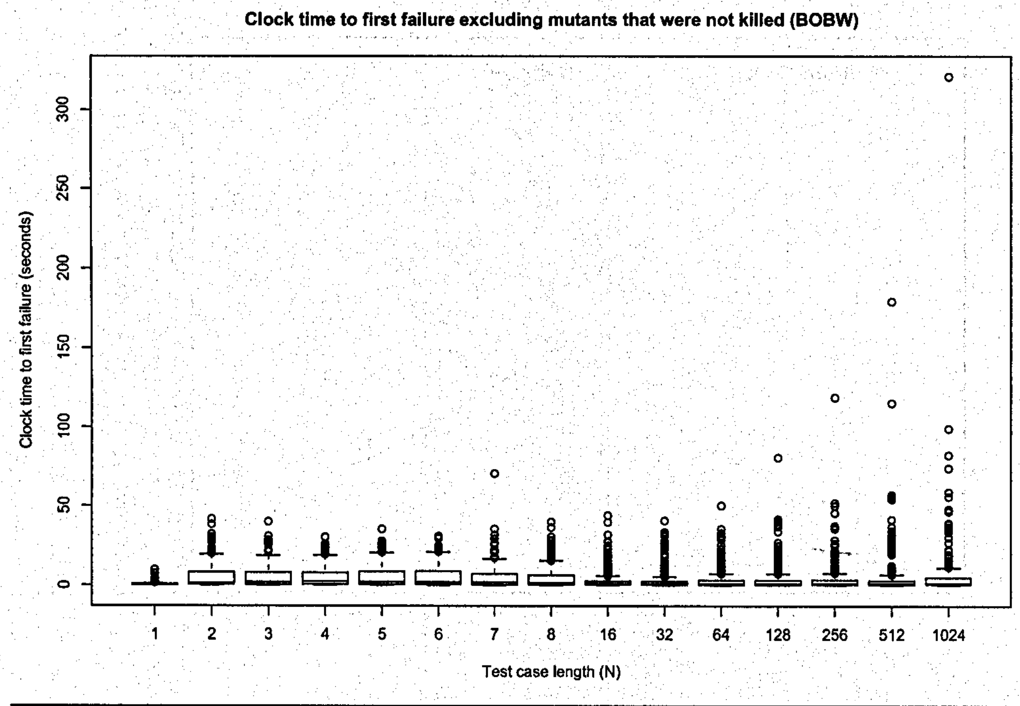
Figure 4.6 Clock time to first failure found by R excluding mutants that were not killed



The experiments have recorded clock time to first failure found by R and BOBW, excluding mutants that were not killed. Figure 4.6 illustrates the clock time to first failure found by R (excluding mutants that were not killed) using a box plot. For R, the clock time to find first failure increases significantly as the test case length increases, and the maximum clock time to first failure is over 3000 seconds.

Figure 4.7 shows the clock time to first failure found by BOBW (excluding mutants that were not killed) using a box plot. For BOBW, the clock time to find first failure does not significantly increase as the test case length increases, and the maximum clock time to first failure is over 300 seconds, but this is much less than the maximum clock time of R. This means when a failure is found, our implementation of BOBW

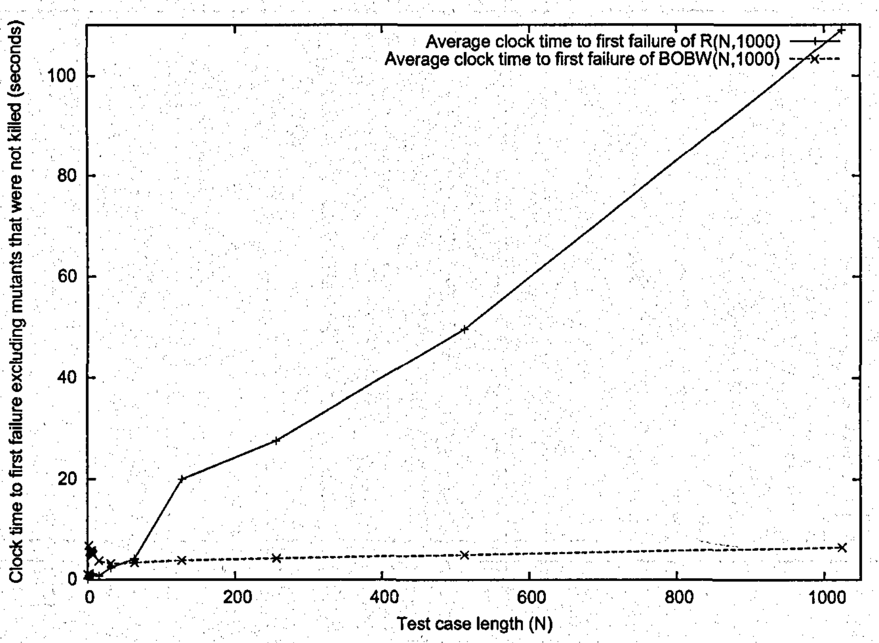
Figure 4.7 Clock time to first failure found by BOBW excluding mutants that were not killed



is faster than R in terms of the clock time to first failure.

Figure 4.8 depicts the comparison between R and BOBW in terms of the average clock time to first failure (excluding mutants that were not found) using a line graph. It clearly shows clock time to first failure increases more for R than for BOBW, and that the average clock time by BOBW is much less than the clock time by R at almost all times. We should note here that figure 4.8 has excluded mutants that were not killed. As discussed before, the number of mutants killed by our implementation of BOBW is less than the number of mutants killed by R, and it remains almost unchanged since test case length 16, so the figure is (probably) showing average clock time to first failure just for the mutants that can be killed at small test case lengths.

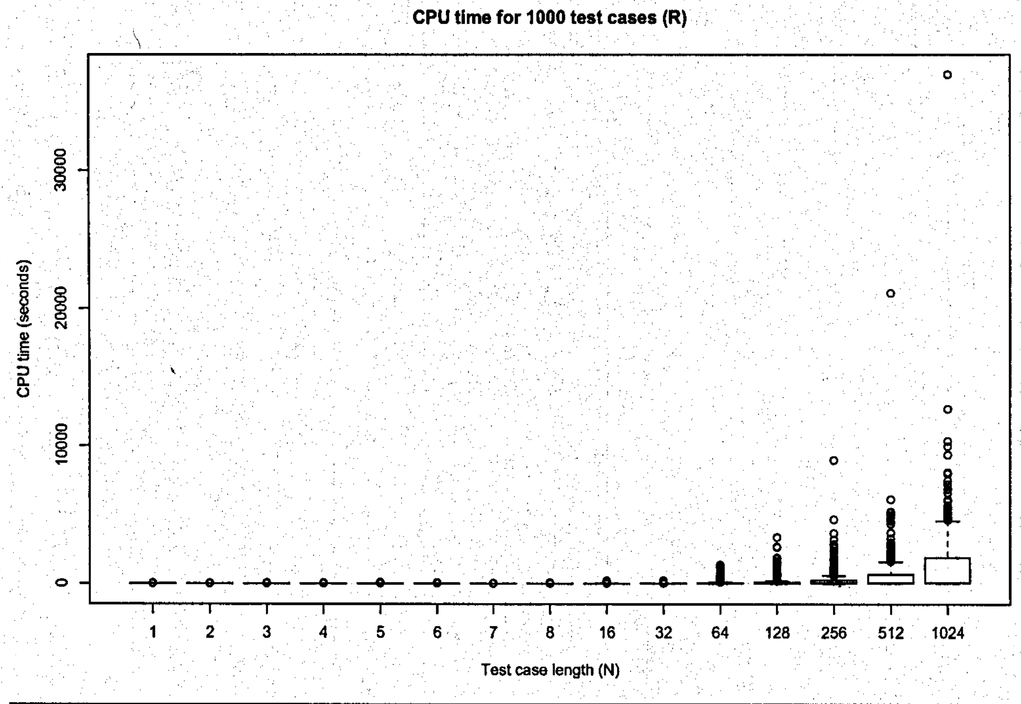
Figure 4.8 Comparison of average clock time to first failure found by R and BOBW excluding mutants that were not killed



Another aspect studied in the experiment is the total amount of CPU time taken by runs in the phase 2. One of our interests is the number of failures found per CPU second. BE achieves its *highest* number of failures per CPU second which is 0.0014 at $n = 2$, and decreases consistently to as low as 0.00018 at $n = 8$. In comparison, R achieves its *lowest* number of failures per CPU second (3.70) at $n = 1$. By $n = 8$, where the comparison with BE ends, it achieves 15.44 failures per CPU second.

The experiments recorded the CPU time for a complete run of each mutant. We have calculated the CPU time for 1000 test cases and drawn box plots for R and

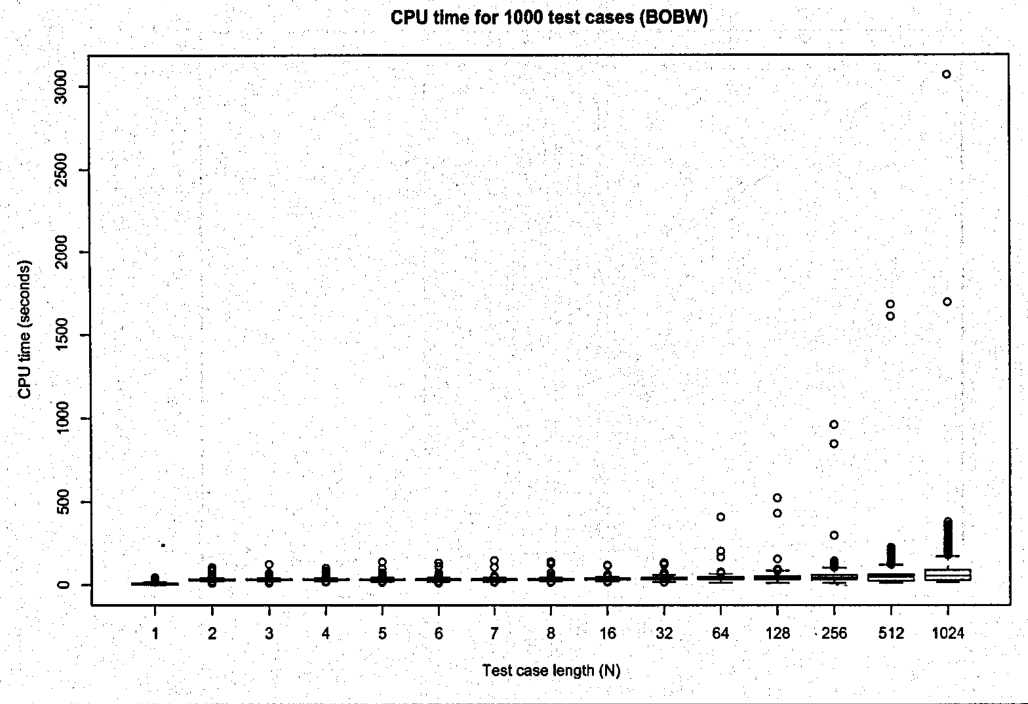
Figure 4.9 CPU time for 1000 test cases for R



BOBW. Figure 4.9 shows the CPU time for 1000 test cases for R using a box plot. The CPU time for 1000 test cases for R increases as the test case length increases. The maximum CPU time taken by a complete run of a mutant is over 30000 seconds which is equal to 500 minutes. It happens when the test case length is 1024.

Figure 4.10 shows the CPU time for 1000 test cases for BOBW using a box plot. Unlike R, the CPU time for 1000 test cases for BOBW does not increase as much as the test case length increases. The maximum CPU time taken by a complete run of a mutant is over 3000 seconds which is equal to 50 minutes. It happens when the test case length is 1024. It is about 10 times less than the maximum CPU time of R.

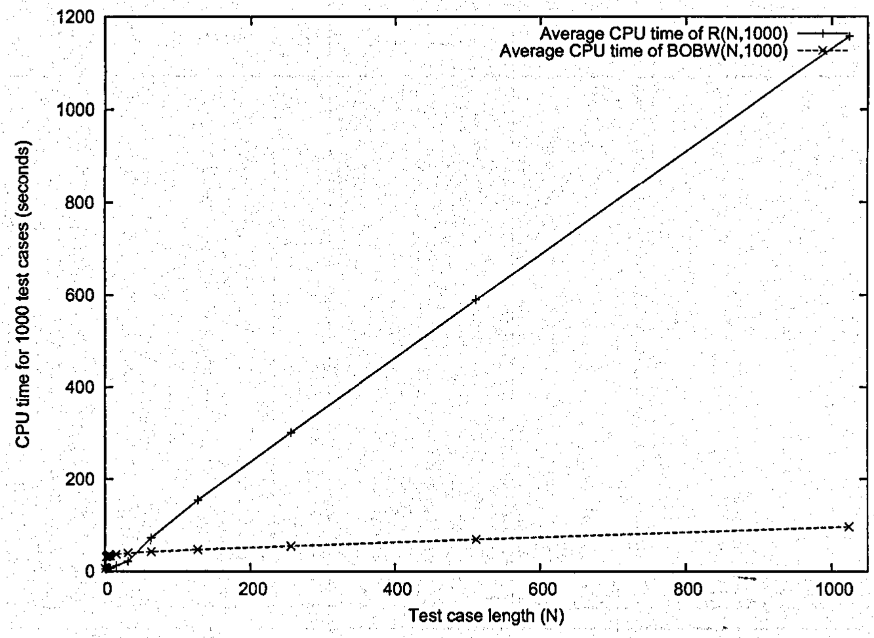
Figure 4.10 CPU time for 1000 test cases for BOBW



We also compare the average CPU time for 1000 test cases run by R and BOBW. Figure 4.11 illustrates the comparison between R and BOBW in terms of the average CPU time for 1000 test cases, using a line graph. The average CPU time of R(N,1000) consistently increases as the test case length increases. On the other hand, the average CPU time of BOBW(N,1000) consistently increases as the test case length increases as well, but it is increasing slightly compared with R. In addition, the average CPU time of R(N,1000) is less than the average CPU time of BOBW(N,1000) only when the test case lengths are short (1-8, 16, and 32). For longer test case lengths, BOBW is much faster than R.

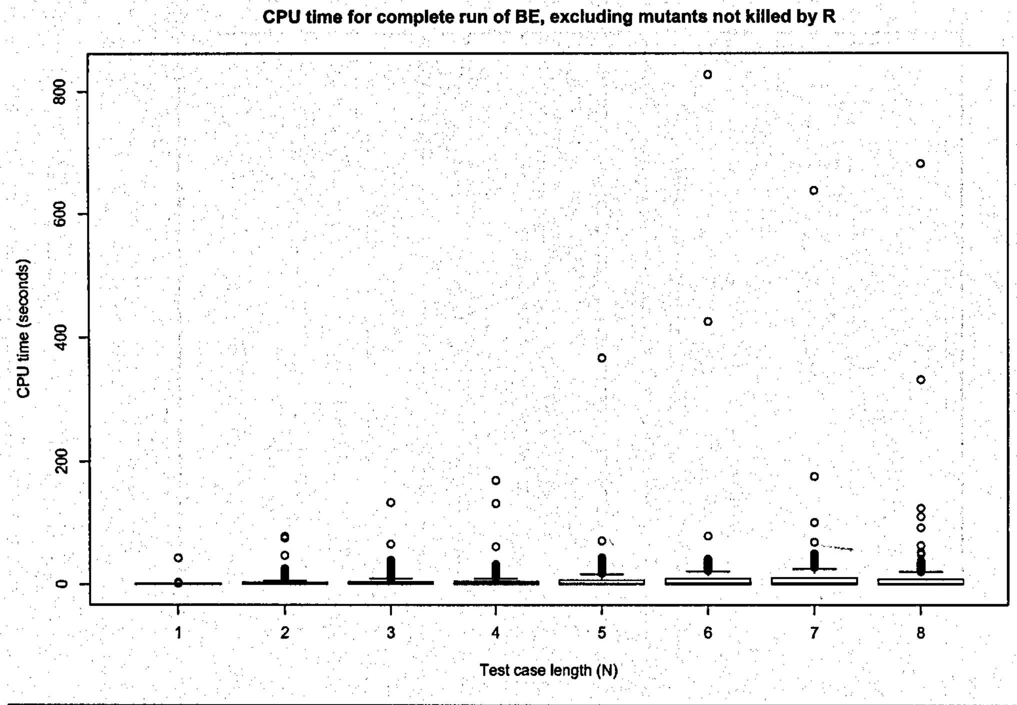
For BE, we use a different approach to do the experiments because it is infeasible

Figure 4.11 CPU time for 1000 test cases comparing R and BOBW



to run a complete set of test cases of BE due to its large search space. We run BE testing on subject units until a failure is found or $E(n)$ test cases have been run. From test case length 1 to 8, we recorded the CPU time for a complete run of BE, excluding mutants that were not killed by R. Figure 4.12 illustrates the CPU time for complete run of BE excluding mutants that were not killed by R using a box plot. An interesting thing in the figure is that the maximum CPU time for complete run of BE happens when the test case length is 6, not 8. It may contradict our intuition that increasing test case length should increase the CPU time taken by a run of BE. There is an explanation to the contradiction. The maximum CPU time taken by a run of BE happens when it executes test cases on mutant 17 of the `java.util.Vector` class. The CPU time taken by a run of BE is calculated by multiplying the CPU time taken by each test case by $E(1)$. Therefore, the reason that it takes the most amount of

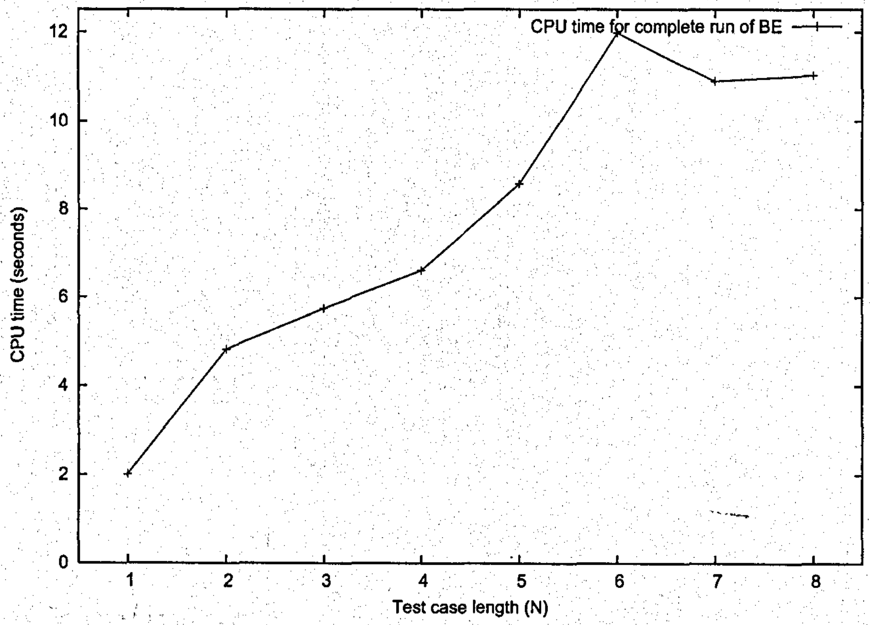
Figure 4.12 Box plot for CPU time for complete run of BE, excluding mutants that were not killed by R



time to finish is because the result of multiplying time taken by each test case by the number $E(1)$ of R(6,1000) for mutant 17 of `java.util.Vector` class is the largest among all subject units. The number $E(1)$ of R(6,1000) is 842, which means if BE cannot kill the mutant 17 of `java.util.Vector` class at test case length 6, it needs to run all 842 test cases until it detects a failure. According to the calculation above, we can tell the maximum CPU time taken by a run of BE happens when it executes test cases on mutant 17 of `java.util.Vector` class at test case length 6.

Figure 4.13 shows a line graph of the CPU time for a complete run of BE, excluding mutants that were not killed by R. Generally speaking, the average CPU time for a complete run of BE consistently increases as the test case length increases. The

Figure 4.13 Line graph for average CPU time for complete run of BE, excluding mutants that were not killed by R



maximum average CPU time for a complete run of BE is about 12 seconds which occurs when the test case length is 6. As discussed above for figure 4.12, we can reasonably explain why the peak happens at test case length 6, not 8.

Chapter 5

Conclusion

5.1 Conclusion

This thesis has closely examined different automated unit testing strategies. It first introduces three automated unit testing strategies: bounded exhaustive, randomized, and best-of-both-worlds. Then it describes the necessity for comparing these three strategies. Based on Andrews et al.'s approach which provides a mechanism to precisely compare the strategies, a test program named *Universal Test* has been developed for implementing the strategies and running experiments, and the details of the implementation have been discussed. Several experiments have been conducted, and experimental data has been collected to figure out the effectiveness and efficiency of the strategies and how increasing test case length affects the failure density. According to the experimental data, this thesis has shown that the failure density increases as increasing the test case length, and randomized testing strategy is more effective than bounded exhaustive testing strategy on average cases.

More specifically, this thesis concludes the following points. First of all, randomized testing is able to find failures in less time and fewer number of test cases than (naive) bounded exhaustive testing, unless failure densities are low. Second, failure densities can be increased by increasing test case lengths, which partly jeopardizes the effectiveness of bounded exhaustive testing. Third, this thesis introduces an explorative testing strategy, named "best-of-both-worlds", which combines both bounded exhaustive and randomized testing strategies. The combined strategy should take advantages of both bounded exhaustive and randomized strategies. Although the best-of-both-worlds strategy has not achieved the results regarding the ability to find failures as we expected, the experimental data shows the best-of-both-worlds strategy is efficient, in terms of the CPU time used to find failures and the clock time to find the first failure.

This thesis only concentrates on comparing the naive bounded exhaustive strategy with the general randomized strategy. Therefore, the experimental results do not resolve the question of whether some optimized implementations of bounded exhaustive strategy would outperform some particular, optimized implementations of randomized strategy on particular subject units or even the same subject units as used in this thesis. However, the experimental results more precisely answer the question of how, when and why randomized strategies can be useful in unit testing. This conclusion may be helpful for people implementing model checkers and other testing tools using randomness or randomized testing strategies.

This thesis also presented the test program development and discussed certain problems encountered during the design and implementation. The design of the test program plays an important role because a good design provides us with flexibility to implement other explorative testing strategies or particular, optimized BE and R strategies painlessly. This thesis has also discussed the reason that BOBW could not beat R regarding the abilities of finding failures. Based on theoretical analysis by Andrews et al., BOBW is more effective and efficient than BE and R. However, our experiments have only shown the efficient side. The implementation of BOBW highly impacts the effectiveness in finding failures.

5.2 Future Work

This thesis has presented an empirical study on comparing different automated unit testing strategies in a formal manner. This thesis provides software testers an insight on the relationship among bounded exhaustive, randomized and combined strategies. Software testers can better understand and estimate how, when and why randomized strategies can be useful in unit testing. With the help of this thesis, testers can design their test suites or test cases in more effective and efficient ways by increasing the test case length or using different testing strategies.

Several improvements can be made to the test program and experiments. Due to the time limitation, we have only applied three testing strategies to and run the experiments on the `java.util` classes. Since the underlying algorithms of implementing

BE, R and BOBW are general, we can apply similar algorithms to other subject units in more languages. One possibility would be applying the algorithms to `sglib` in the C programming language. `SGLIB` is a simple generic library for the C programming language. It defines useful macros for manipulating common data structures. It provides generic implementation for sorting arrays and manipulating the following data structures:

- linked lists
- sorted linked lists
- double linked lists
- red-black trees
- hashed containers

Manipulating a data structure includes insertion, deletion, search and iterator traversal of elements. `SGLIB` provides a basic set of functions (macros) for manipulating each data structure. It is like the Standard Template Library for the C++ programming language.

Obviously, one drawback of the test program is the implementation of BOBW. The choice of constants in the current implementation of BOBW did not achieve the desired properties of a pseudorandom number generator. We implemented BOBW by choosing a large prime number p and generating the next test case index by adding p modulo z. Without factorizing a large number z, we need to carefully choose p in order to meet the criteria that z is not close to a multiple of p or vice versa. The linear congruential random number generator would be a suitable substitution of the existing implementation.

Another important aspect of future work would be comparing particular and optimized BE strategies with particular and optimized R strategies. This thesis has 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. On the other hand, for very large search spaces, it is often not realistic to perform a complete run of the naive BE strategy. It would be interesting to optimize the BE strategy, such as dividing large search spaces and exploring, to perform complete runs of BE for longer test case lengths. Respectively for R, many optimization techniques can be applied as well. Genetic and heuristic algorithms may optimize the existing randomized testing strategy so that optimized R may become more effective.

References

[1] James H. Andrews. A case study of coverage-checked random data structure testing. *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 316–319, Sep 2004.

[2] James H. Andrews, Lionel C. Briand, and Yvan 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.

[3] James H. Andrews, Felix Chun Hang Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007.

[4] James H. Andrews, Yihao Zhang, and Alex Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, December 2010.

[5] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[6] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[7] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin J. Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–339, April 2005.

[8] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[9] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[10] Edward Kit. *Software Testing in the Real World: improving the process.* Addison-Wesley Publishing Company, Inc., 1995.

[11] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT Computer Science and Artificial Intelligence Laboratory, September 2003.

[12] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of Java programs. In *16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov 2001.

[13] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[14] Glenford J. Myers. *The Art of Software Testing.* Wiley, New York, 1979.

[15] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, January 1992.

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

[17] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[18] Willem Visser, Corina S. Păsăreanu, and Radek 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.