Spring 2-4-2013

Test Advising Framework

Yurong Wang
University of Nebraska-Lincoln, lucymagic.unl@gmail.com

TEST ADVISING FRAMEWORK

by

Yurong Wang

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Matthew Dwyer and Sebastian Elbaum

Lincoln, Nebraska

January, 2013

TEST ADVISING FRAMEWORK

Yurong Wang, M. S.

University of Nebraska, 2013

Advisors: Matthew Dwyer and Sebastian Elbaum

Test cases are represented in various formats depending on the process, the technique or the tool used to generate the tests. While different test case representations are necessary, this diversity challenges us in comparing test cases and leveraging strengths among them - a common test representation will help.

In this thesis, we define a new Test Case Language (TCL) that can be used to represent test cases that vary in structure and are generated by multiple test generation frameworks. We also present a methodology for transforming test cases of varying representations into a common format where they can be matched and analyzed. With the common representation in our test case description language, we define five advice functions to leverage the testing strength from one type of tests to improve the effectiveness of other type(s) of tests. These advice functions analyze test input values, method call sequences, or test oracles of one source test suite to derive advice, and utilize the advice to amplify the effectiveness of an original test suite. Our assessment shows that the amplified test suite derived from the advice functions has improved values in terms of code coverage and mutant kill score compared to the original test suite before the advice functions applied.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisers, Dr. Matthew Dwyer and Dr. Sebastian Elbaum. This work could not have been accomplished without their support and guidance. Dr. Dwyer is very sincere and encouraging and Dr. Elbaum is so patient and understanding. I want to thank them for the great effort they put in to guide me through my research path from a very basic level. I am deeply grateful for the great opportunity to work both them.

I also want to thank Dr. Suzette Person for all she's done to help me with this work. She is so nice, thoughtful and always ready to help when I have a problem. I would also like to thank my committee member, Dr. Myra Cohen for offering great suggestions and taking time to read this thesis.

Friends in ESQuaReD helped so much during my course of study. I thank them for the fruitful discussion on research ideas, the help to tackle a technical issue, and all the small talks among girls. I also want to thank Shelley Everett, Deb Heckens, Charles Daniel and Shea Svoboda for their generous help on the administrative tasks.

Finally, I want to thank my family for their continuous support throughout my course of study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Tests come in many forms, they have varied goals, and they encode valuable information from different stakeholders and development artifacts. A requirements engineer may carefully specify a set of acceptance tests for a system during its initial inception. A developer implementing a tricky algorithm may write some throw-away tests to exercise behavior she is not confident in. A test engineer may seek to craft a suite of tests that provide comprehensive coverage of a design model or specification [68, 48] or implementation [80]. A system engineer may develop end-to-end tests to expose system performance or resource consumption [20]. When viewed broadly, the space of *all* such tests constitute a rich body of information about how a system should function.

When performed manually, such testing activities result in test descriptions that detail how to establish a state, i.e., the *test input*, in which the *system under test* (SUT) is executed and how the resulting state should be judged, i.e., the *test oracle* [67]. Tests might be expressed using domain specific languages [71], as code using special libraries [7], or using custom encodings of input and output data [23]. While this diversity may enable different stakeholders to contribute to the overall testing effort, *differences in test description present an obstacle to extracting information from a population of tests*.

Automated test case generation techniques can provide support to some of those efforts. Given the definition of system inputs, random generation of values has proven effective for certain systems, e.g., [53, 52, 76, 5, 14, 29, 34, 81]. Analysis of program source code can be applied to generate tests that cover unlikely input scenarios, e.g., [63, 73, 79]. If design models or specifications are available, tests can be generated that exercise a system based on that design [9, 38]. Such test generators provide an increasingly important capability that complements manual testing. While automated test generators can produce tests that humans might not produce or do so more efficiently, *limitations in test generator interfaces present an obstacle to customizing the test generation process.*

We hypothesize that information captured in different types of tests and test generators is *complementary*. Furthermore, if the highlighted obstacles are overcome, then that information could be extracted and applied in order to generate more effective tests. In this thesis, we explore an approach that transforms diverse tests into a common encoding, extracts information from those test encodings, and applies that information to either directly improve existing tests or to enhance the capabilities of test generators.

1.1 Motivating Example

Consider the code in Figure 1.1 which contains excerpts from a Binary Search Tree class with a series of methods including `insert`, `delete`, and `balance`, and the three kinds of tests that exercise that class in Figures 3-5. These tests come from different sources, have different representations, and present different tradeoffs.

The test in Figure 1.2, was designed by a tester and implemented in some scripting language. The test generates, through a series of insertions with deliberately chosen values, a tree with nodes only on the right, a corner case of interest to the tester. More interesting is that the test includes not only a check on the tree height against a constant, but also a more

```
public class Binarytree {
    static class Node {
        Node left; Node right; int value;
        public Node(int value) {
            this.value = value;
    }   }
    public void insert(Node node, int value) {
        if (value < node.value) {
            if (node.left != null) {
                insert(node.left, value);
            } else node.left = new Node(value);
        } else if (value > node.value) {
            if (node.right != null) {
        ...
    }  }
    public void delete(Node node, int value) {...}

    public void balance(Node node) {...}
}
```

Figure 1.1: SUT

general check that uses an invariant that characterizes the relationship between a node and its children (this invariant is reused across many of the hand-coded tests).

```
create BST root with value 0;
insert node in BST with value 1;
insert node in BST with value 2;
insert node in BST with value 3;
insert node in BST with value 4;

check(BST height = 4);

For each node in BST
  if Left node exists then
    check (Left node value < Node value)
  if Right node exists then
    check (Right node value  >= Node value)
```

Figure 1.2: Developer's Test in Scripting Language

The tests in Figure 1.3 were generated by an automated tool like Randoop [53] that, given a class, will invoke its public methods using a set of predefined values or values returned by previous method calls, and use some form of regression oracle. The tool is somewhat limited by its pool of default values(e.g, 0,1, -1, 10, 100 for integer values), and limited by the primitive nature of its oracle, but it can quickly explore more and longer sequences than those often implemented by the tester. These sequences can cover a lot of functionality and also lead to the creation of more complex heap structures.

```
public void Test1() throws Throwable{
  Node rootnode = new Node(0);
  insert(rootnode, -1);
  insert(rootnode, 1);
  insert(rootnode, 10);
  insert(rootnode, 100);
  balance(rootnode);
  assertTrue(rootnode.value, 0);
}
public void Test2() throws Throwable{
  Node rootnode = new Node(0);
  insert(rootnode, 1);
  insert(rootnode, -1);
  insert(rootnode, 10);
  insert(rootnode, 100);
  delete(rootnode, -1);
  delete(rootnode, 100);
  delete(rootnode, 0);
  assertTrue(rootnode.value, 1);
}
```

Figure 1.3: Test generated with Randoop

Figure 1.4 introduces a more abstract form of tests that represents partitions in the input space of the `insert` method, each leading to the execution of a different path in the method. This kind of test, often produced by tools that operate based on some form of symbolic execution like SPF [73], is concretized by solving the generated constraints. So, for example, solving PC_2 will lead to the creation of a new node to the left of the single existing node. The most valuable part of these tests is that each test covers a unique path through the code and (ideally) all paths are covered. The exhaustive nature of the test suite leads to the discovery of faults that may be missed by the other approaches (PC_1 detect a fault when trying to `insert` on an empty tree).

```
// target insert method
PC_1: node = null

PC_2: node.value == 0 &&
      node.left = null &&
      node.right = null &&
      value < node.value

PC_3: node.value == 0 &&
      node.left = null &&
      node.right = null &&
      value == node.value

PC_4: node.value == 0 &&
      node.left = null &&
      node.right = null &&
      value > node.value

...
```

Figure 1.4: Test “specs” generated with SPF

Although the code excerpt and the test sample is small, it illustrates the diversity we find across tests generated with different goals, information, and tools. What is perhaps less evident given the diversity of formats, are the opportunities to improve these tests by leveraging each other.

Consider the list of exhaustive tests generated by SPF from which we showed just three in Figure 1.4. Although these tests may in practice cover most paths, they can only expose faults that raise exceptions. The lack of a more strict oracle often renders these tests unable to detect many faults residing in paths that are traversed. For example, the lack of an oracle associated with PC_3 means that when attempting to insert a node with an existing value and the node is not created, the test does not report a fault. Now, if the invariant from the manual test in Figure 1.2 would be incorporated at the end of the test instantiated from

PC_3, then the fault would be found. In this case, we say that an **oracle advice** from the manual test could benefit the SPF test.

Now consider the Randoop test in Figure 1.3. These tests are limited by the pool of integer values that the tool uses to invoke the `insert` method. Since the code is not adding new nodes when values already exist in the tree, Randoop cannot generate trees with more than 5 elements, which we assume would be necessary to reveal a fault in the `balance` method. This problem could be overcome if the values from the SPF instantiated tests, which cover all paths in `insert`, or the values used by the tester in the scripted test are used to enrich the initial pool of integer values of Randoop. We call this an **input advice**, which in this case would help Randoop to generate more and richer sequences, leading to complex tree structures.

Last, consider the rich sequences that are explored by the Randoop tests in Figure 1.3. In the second test, a single sequence builds a tree and explores three different removals (of the left node, right node, and root node). Generating a similar test sequence with a tool like SPF would be extremely costly (in the order 5 hours for sequences of length 5) as it would require the exploration of large program space until a tree of that size could be built. In this case, the SPF tests could leverage such sequence to act as a test setup. Similarly, we note that tests developed by the tester could provide valuable sequences for Randoop and SPF tests alike. In this case the sequence consists of just creating the root node, but it is easy to imagine more complex setups such as those involving method calls that follow specific protocols where automatic test case generation mechanisms would struggle. Such cases are instances of what we call a **sequence advice**.

1.2 Research Contributions

In this thesis, we begin to address the limitations that are introduced by diversity in test representations by defining a new test case language to establish a common representation for test cases generated by multiple techniques and by illustrate how the common representation can be used to create synergy between testing techniques. We first show how our test case language can be used to represent test cases that vary in structure and are generated by multiple test generation frameworks. We also present a methodology for automating the transformation of test cases in varying representations into a common format where they can be matched and analyzed. We then illustrate how the strengths of one test suite, stored in the common format, can be leveraged to help improve the effectiveness of other types of test through the presentation of four test advice functions. These advice functions analyze test input values, method call sequences, and test oracles for a source test suite to derive and utilize the advice to amplify the effectiveness of another test suite. Our assessment shows that the amplified test suite derived from the advice functions has improved values in terms of code coverage and mutant kill score compared to the original test suite before the advice functions applied.

The main contributions of our work are:

- We define a test case language (TCL) for representing test cases in a canonical representation. The grammar captures the three components of a test case: Invoke action, Write action and Decision action. We demonstrate the applicability of the grammar through a series of examples covering test cases of varying formats.
- We present a methodology to automatically transform test cases into the grammar and implement the transformation process for three different types of tests.

- We define four advice functions to operate on the common test representation of test suites. Each advice function is evaluated and results show that the advice functions do bring value to the amplified test suite by increasing the code coverage or mutant kill score.

1.3 Thesis Overview

The rest of this thesis is organized as follows. We discuss in Chapter 2 the related work in software testing functional models, test representations, other advising frameworks, approaches and tools for test generation and evaluation. In Chapter 3, we define the test case language used for the common representation of tests and demonstrate the applicability of the test case language through a series of examples. We define four advice functions in Chapter 4 operating on various types of tests represented in the canonical representation and Chapter 5 performs evaluation on each of the four advice functions. Finally, in Chapter 6 we conclude the thesis.

Special Thanks A significant part of this thesis material comes from a conference paper submission with Dr. Suzette Person, Dr. Sebastian Elbaum, and Dr. Matthew Dwyer. I greatly appreciate the tremendous effort they have put in for this piece of work and the paper submission. There could have never been a paper without their generous support.

Chapter 2

Related Work

In this chapter, we discuss several avenues of work related to our test advising framework. We describe related work in theoretical models of testing, the main test generation approaches and tools, alternative test case representations, other existing advising frameworks, and the evaluation tools utilized in our work.

2.1 Theoretical Models of Testing

Informal testing models have existed since the first program was tested, but early work by the testing community recognized the need of more formal testing models to more precisely describe the testing problem and its key components. In 1975, Goodenough and Gerhart first attempted to more formally define the software testing problem by providing a theoretical foundation for testing, and characterizing a test data selection strategy [30]. In their theory, the definition of an ideal test is dependent on the program F and the input domain D. Let F be a program, D be its input domain, $OK(d)$ denoting the result of executing F with input $d \in D$, T be a subset of D, T is considered as an ideal test if $OK(t), \forall t \in T \Rightarrow OK(d), \forall d \in D$. T is successful iff $\forall t \in T \Rightarrow OK(d), \forall d \in D$. Two properties (*reliable* and *valid*) were brought up to help define the test selection criteria. A criteria C is considered as ideal if it is both *reliable* and *valid*. By *reliable*, it means the tests selected by C have consistent results (they all pass or they all fail). By *valid*, it means the tests selected by C can reveal the error if there is one in the program F.

Later on, both the work by Weyuker and Ostrand [77] and the theory by Gourlay [31] refined Goodenough and Gerhart's framework. Weyuker and Ostrand brought up the concepts of a uniformly ideal test selection criteria and a revealing subdomain. A criteria C is considered uniformly ideal if C is both *uniformly valid* and *uniformly reliable*. Goodenough and Gerhart defined the concepts of *valid* and *reliable*. And a test selection criteria C is considered *uniformly valid* if C is *valid* for all programs and considered *uniformly reliable* if C is *reliable* for all programs. This is to address the flaw in Goodenough and Gerhart's work of having the definition of *valid* and *reliable* restricted to the scope of a single program and specification. A subdomain S is a subset of the input domain D. A subdomain S is revealing if the output of an input in S is incorrect, then all test sets which satisfy C is also incorrect.

In Gourlay's theory, a testing system is defined with respect to a set of arbitrary programs, specifications and tests. Let P be the set of all programs, S be the set of all specifications, T be the set of all tests, $p\ ok_t\ s$ denoting that test t performed on program p is judged successful by specification s, $p\ corr\ s$ denoting the fact that p is correct with respect to s, a testing system is defined as a collection of $< P, S, T, corr, ok >$, where $corr \subseteq P \times S$, $ok \subseteq T \times P \times S$, and $\forall p\ \forall s\ \forall t\ (p\ corr\ s \Rightarrow\ p\ ok_t\ s)$.

Staats et al. in [67] extended Gourlay's Framework to include oracles into the mix. They argue that the important oracle aspect of testing has been largely overlooked in all previous foundational work. Staats et al. refined Gourlay's Framework with two major changes. They replaced the predicate ok with the set of O of test oracles, and also added a predicate defining correctness of tests in T. A testing system is defined as a collection of

$< P, S, T, O, corr, corr_t >$, where $corr \subseteq P \times S$, $corr_t \subseteq T \times P \times S$, and $\forall p \in P,\ \forall s \in S,\ corr(p, s) \Rightarrow \forall t \in T\ corr_t(t, p, s)$.

All the approaches listed above provide fundamental support for the research work in software testing. However, they all define a test case based on other artifacts like the programs and the specifications. This can be problematic since independent of the SUT, we cannot compare tests or use tests based on these definitions. In the work [30, 31, 77], a test is considered as data and as an entity related to programs and specifications, and in [67] a test is defined as a sequence of inputs accepted by some program. We argue that all these functional models are too vague and do not specify the components a test. We will explore a novel approach in Chapter 3 which focuses on the tests, exploring the fundamental elements in the tests independent of a particular implementation of the SUT. This provides us a clearer view of what the tests are actually doing, and leads to opportunities to enhance the elements involved in the tests.

2.2 Test generation Approaches and Tools

There are two directions for software testing - automated and manual testing [40]. We discuss in this section both manual testing and two alternatives for automated testing.

Manual Approach Even though it is considered labor intensive and expensive [22, 41], manual testing is still a well-known and widely used testing technique. In manual testing, programmers embed their understanding of the SUT inside the tests they write, e.g., what values to use for method invocation, which methods to call or which attributes to check after the test execution. This knowledge is hard for the automated techniques to obtain, and also makes the manual tests valuable to have.

Random Approach In random testing, test cases are sampled from the input domain randomly [37]. Although random testing is considered as a naïve approach [50], it has successfully detected defects in some widely used applications like Unix utilities [47], Haskell programs [15]. Random testing tools are easier to implement and can generate a large number of tests within a short period of time. However, being entirely random, they can produce many illegal or equivalent test inputs. They cannot guarantee any coverage goal while exercising a random sample of the total input space, and they may also be challenged to cover sections of code guarded by predicates that cannot be easily traversed through random values. Many tools have been developed based on the idea of random testing (e.g., [5, 14, 17, 18, 25, 29, 53, 65]). Among these, Randoop [53] is the one that we use in this thesis. It utilizes a feedback-directed random test generation technique. Randoop generates unit tests by generating sequences of methods and constructors for the SUT, executing the generated sequences and discarding any illegal sequences. Studies have shown Randoop is capable of generating cost-effective tests [53].

Systematic Approach Systematic testing techniques generate test inputs up to a size bound. Different from random testing, systematic testing enumerates all inputs within the given bound deterministically. Being systematic and exhaustive, it has consistent coverage of the SUT and ensures the correctness of the input space explored. However, systematic testing suffers from the lack of scalability and tends to focus on a part of the state space with less diverse test inputs. One alternative to systematic testing is a symbolic-execution based approach. In symbolic execution, path constraints are collected based on symbolic inputs used for method parameters. Concrete input parameters derived through solving the path constraints are in turn used to create concrete test cases. It has been implemented in tools like JPF [72], Symstra [79], and XRT [33], jCUTE [63]. JPF is widely used to test mission critical applications at NASA. In Chapter 4 and Chapter 5, we use SPF [55, 56] for test generation in the systematic fashion.

Test cases generated through different approaches have their own strengths and weaknesses. In Chapter 4 and Chapter 5, we explore tests generated with all 3 different approaches listed above. With the understanding of the advantages and disadvantages of each type of tests, we leverage the information from one type of tests to provide advice to other type(s) of tests.

2.3 Test Case Representations

Tests can take on various forms depending on the process, the technique or the tool used to generate the tests. We discuss in this section a few of the most commonly used test representations and how they are used or generated in testing software.

Category Partition Method The category-partition method [51] is a specification-based functional testing technique. It can derive test specification based on analysis of the functional specification of the SUT, and tests in textual descriptions can also be transformed from the test specification. The generated tests require manual effort to be executed and only specify the potential test inputs and setups. Ostrand et al. in [4] attempted to automate the testing process and added test results into the test representation. A test case now include both test input and expected results. Although subsequent research effort in the literature built on the category partition method (e.g., [4, 12, 44]), there is no uniform representation for the tests generated. For example, test inputs can be parameter values or scripts running Unix commands, etc.

Tests in xUnit Framework xUnit refers to a group of well-known testing frameworks. The concrete instances include JUnit for Java, CppUnit for C++, NUnit for .NET, etc. A number of basic components are shared among all xUnit frameworks with varying implementation details. For example, they all have the component of *test suite* which is composed of a number of *test cases*, and also the *assertion* component is used by all frameworks to verify the execution results of tests. The xUnit frameworks have made it easy for the automated test generation process. For example, a number of tools output test suites in JUnit framework like JCrasher [17], Randoop [53], JPF-symbc [55, 56] with *SymbolicSequenceListener*, and Symstra [79]. The tests in xUnit frameworks are rather standard. They are represented as source code which exercise the implementation of the SUT. However, it is not clearly stated what a test is composed of. For example, we can only infer that the *assertion* statements are the oracles in the tests and all other statements are used to set up the test input.

Tests in Scripting Languages A test script has a list of instructions which exercises the functions in the SUT to check whether they are implemented correctly. For example, the test input generated by category partition method we discussed above can be specified in shell scripts running Unix commands. Test scripts are also often used to test GUI-based applications (e.g., [11, 27, 32]). GUI testing is considered tedious and labor intensive. Thus test scripts are developed to help automate the process and can be written in scripting languages like JavaScript or VBScript. There are also commercial software testing automation tools available for composing test scripts like HP Quick Test Professional (QTP) and Borland SilkTest. The instructions contained in the test script captures in general the actions to perform on the SUT and the expected results after performing the specified actions. It is rather similar to the tests in xUnit frameworks in terms of the representation, but all written in different languages.

JMLUnit Tests JMLUnit [13] has its test cases represented as part of the program specification written in JML [43]. JML is a formal language specification which can define invariants for classes, pre- and post-conditions for methods. JMLUnit takes advantage of the in-line JML specification developed for the SUT and performs testing at runtime by checking concrete executions against the in-line specifications. Tests, in this case, are instrumented into the implementation of the SUT based on the rich inline specification. Thus it is hard to identify what the test is composed of and hard for other techniques to communicate with the JMLUnit tests.

Tests As Constraints This is an abstract type of tests, in which the input values are specified as constraints. SPF [55, 56] generates tests represented in path constraints with each path constraint represents an execution path of the SUT. Schulte et al. [70] also works at the abstract level with path constraints to generate tests. Each abstract test is represented as a path constraint generated, which captures the relationship constraints that parameters involved need to satisfy.

Parameterized Unit Test In parametric test cases, parameters are allowed in the tests in order to be more general and expressive. Different input values can be assigned leading the test case on a different path of the program. A number of research effort has integrated with the concept of parameterized unit test. The commonly used JUnit framework starts to support parameterized test with @*RunWith*, @*Parameter* and @*Parameters* annotations. Schulte et al. [70] works with parameterized unit tests and path constraints for method under test to generate concrete input values for tests. The framework implemented by Zeller et al. [26] supports the generation of parameterized unit tests.

Ad-hoc Under many occasions, every practitioner may have their own adaptation to better fit their domain to skills. Differential Unit Tests (DUT) are represented as a pair (pre-state, post-state) of XML files in [23]. The pre-state of the DUT captures the program state before the method invocation and the post-state captures the program state after. Systems built in [6, 66] target the database systems and thus output tests in SQL queries. Siena is a Scalable Internet Event Notification Architecture [10] and the tests available in the Subject Infrastructure Repository (SIR) [19, 60] are represented in shell scripts. Each specific kind here takes on its own representation suitable to its own domain.

Given the variety of test representations used in the literature, we argue in Chapter 3 about the benefits of unifying the representations in testing literature and present how test cases in various formats can be represented uniformly and the benefits of having the same canonical representation.

2.4 Test Advice

Even though not explicitly called advice, researchers in related work aim to improve a technique or a test generation tool by leveraging information from other sources. Given a test, we can extract three different and important pieces of advice including the method input arguments used to invoke methods, the method sequence used to set up the program state, and the oracles utilized to check the correctness of the program state. We will discuss recent research work on each aspect.

Method Input Values We identified a few pieces of work, which involves getting "better" tests through better input selection. Robinson et al. collected the input values used in the source code to guide a random test generation tool to get a richer set of tests [59]. In a different direction, in PUT (Parametrized Unit Test), parameterizable arguments are allowed in test methods in order to improve expressiveness [70]. Often, symbolic execution and constraint solving are utilized to provide "good" input values for the parametrized test methods.

Test Method Sequence Palulu [3] infers a call sequence model from a sample program execution and use it to guide the test generation process. Palus [82] enriches the call sequence model with method argument constraints and also take advice from method dependencies extracted from a static analysis of the SUT, in order to guide the subsequent test generation process. MSeqGen [69] mines the available source code from open source projects to get method sequences which are considered useful for random or systematic test generation tools. The difference between the two approaches is that the method sequence advice used in MSeqGen includes both the list of methods to be invoked and the input arguments used for these method invocation. Palus abstracts the method sequence models from the sample code and only retrieving the advice on the list of methods to be used while input values used for the method calls are still picked randomly at test generation time. RecGen [83] uses a static analysis to identify the relevant methods and uses the information to guide the test generation process.

Test Oracles The oracles used in automated tests have always been a big challenge [8]. There have been a few strategies employed in the literature to address this challenge like *Decision Table*, *Regression Tester* [64]. A number of tools utilize the regression strategy for the oracles used in generated tests [23, 53, 78], where version $N's$ behavior of the SUT is used to validate the behavior of version $N + 1$. Although they all use regression strategy, the tools differ in how they perform it. Randoop checks the variable values which capture the return values of method calls [53]; Carving checks fields [23]; and Orstra checks public method return values [78]. Also, Randoop leverages the API contracts information while generating tests by checking a small set of contracts that the SUT should follow, including *Reflexivity of equality*, and *Symmetry of equality* for example. By default, JPF checks whether the SUT has unhandled exceptions, and catches some generic program properties like *"ArithmeticException: division by zero"*. All these works take advantage of the commonly accepted programming contracts to provide some oracle support in the generated tests. An alternative to this direction is utilized in JMLUnit [13]. JMLUnit leverages information from the formal specification for the SUT and use it as the oracles while performing runtime checks. Landhauser et al. [42] discussed the opportunity of reusing oracles from other tests by cloning, but they also pointed out failing tests can be derived when cloning the oracles.

In a similar fashion to all the directions of advice, in Chapter 4 and Chapter 5 we also explore the potential of advice extracted from an existing type of tests based on the unifying representation of the test cases.

2.5 Test evaluation Tools

There are four major test evaluation directions in the literature - code coverage, mutant kill score, manual fault seeding, and evaluating program with real faults in them which were discovered in later versions. In this section, we focus on the first two since we will use them later in Chapter 5 of the thesis. We discuss in the following sections the most widely used tools for the first two test evaluation metrics.

Structual Coverage Measures Code coverage is a structural software testing metric widely used in software testing to evaluate the effectiveness of a test suite. There are a variety of ways to measure code coverage, e.g., statement coverage, predicate coverage, path coverage, function coverage [50, 58, 80]. The basic intuition behind code coverage is that the code needs to be executed before the bug can be found. And the more code covered through testing, the higher probability of revealing underlying defects. A number of coverage based testing tools have been developed (e.g., [16, 24, 57]). Cobertura [16] is one of the commonly used tool in the evaluation of software testing techniques. Cobertura instruments the Java byte-code after it has been compiled. It displays in a HTML or XML format the statement and predicate coverage for each class, each package and the whole SUT. It also provides detailed information about which lines or predicates are covered or not covered. We use Cobertura as a test evaluation tool in our assessment in Chapter 5.

Mutant Detection Capability Measures In mutation testing, one small change is seeded in the SUT at a time which generates one mutant of the original SUT. Typical changes include replacing a subtraction operator with an addition operator, changing values of constants and literals, removing a method call, etc. After the mutants are generated, all tests are run on the mutated version of the SUT. If at least one test fails, the mutant is said to be killed. If all tests pass, the mutant is said to be alive. Unless all mutants are killed, the test suite is considered inadequate and more enhancement needs to be made. Andrews et al. [2] concluded that mutation testing generates mutants similar to real software defects and can provide trustworthy results. A number of mutation testing tools have been developed for Java (e.g., [39, 45, 49, 61]). Among these, Javalanche [61] is a recent tool which aims to address the efficiency challenge of mutation testing with fully automated support. They attempt to perform efficient mutation testing through manipulating bytecode to avoid recompilation cost, leverage coverage data to execute the subset of tests which exercise the mutant, reduce the probability of equivalent mutants, etc. It is shown in the study [61] that Javalanche is efficient with large programs. In our study, we use Javalanche as the mutation testing tool to evaluate the effectiveness of the test suites.

Chapter 3

A Framework for Representing Tests

Software engineers generate a diversity of tests to validate their software. This diversity manifests along multiple dimensions. Diversity manifests in what type of test is generated. For example, testers require functional tests to validate features and non-functional tests to validate performance attributes. Diversity manifests in the process followed and the techniques employed to generate the tests cases. For example, some testing activities such as input generation may be performed automatically while others may require manual effort to incorporate test oracles. Diversity manifests in when the tests cases are generated, which also may have an impact on their granularity. For example, unit tests may be generated for target classes as they are developed, while system tests may be developed throughout the project and executed towards the end of each complete product cycle. Even the personnel performing testing, with their domain knowledge and preferences, insert a certain level of diversity into the testing process. These different types of testing diversity are necessary to validate complex software from multiple perspectives and at different stages of development.

Yet, for all its strengths, there is an aspect of test diversity that is not only unnecessary but limiting: the diversity in test case representation.

Diversity in representation is limiting in at least two ways. First, the knowledge embedded in the tests is unnecessarily constrained to a specific technique or tool for which it was originally designed. For example, JMLUnit [13] is an automatic unit test generation tool based on JML (Java Modelling Language) specifications. It uses the in-line specification statements written in JML to derive tests performing run-time assertion checking on the SUT. This rich in-line specification, however, cannot be easily communicated to other techniques. Furthermore, with the number of emerging testing techniques and tools, lacking a common representation limits the comparison across techniques. For example, the Differential Unit Test (DUT) [23] carved from system tests are cost effective at testing changed code. However, their custom representation in XML files makes it hard for other techniques to leverage their detailed state information. Other common types of tests include manual tests (which often lack a common format), load tests (a non-functional type), and JPF's abstract tests represented as path constraints [55, 56]. Also, test case representation diversity makes it harder to develop synergy between existing testing approaches, techniques, and tools. In Chapter 4 and Chapter 5 of the thesis, we will illustrate the potential of having such a common test representation, through a family of testing advice that leverage information from one type of tests to improve the coverage or fault detection power of other types of tests.

Second, diversity in representation makes it costly to maintain large and diverse test suites as tests may require very specific support to evolve with the program in order to remain relevant. As shown by Orso et al. [54], test suites rarely remain the same once they are created. Test suites evolve in a number of ways, modifying method arguments, changing method-call sequences, or adding better assertions for example. A uniform test representation will facilitate the test repair techniques as they would not need to consider various representations.

We start this chapter with the definition of a test case representation that is meant to unify the format diversity in tests. Then we explore the applicability of the test representation by looking at different types of tests and how they are transformed into the common test representation format. We also talk about the implementation of the framework and the transformation of various types of tests into the framework. Finally, we discuss the robustness of the current implementation of the framework.

3.1 A Common Test Representation

It is relatively common in the testing literature (e.g., [52, 4, 74]) to define or assume that a test case is a pair, (i, e), of inputs i that are used to exercise the software under test (SUT) and expected outputs e that are compared against the outcome of the SUT to judge its correctness. The functional models of testing discussed in Section 2.1 provide additional informal definitions of a test and how it relates to a program, its specifications, and oracles. However, the definitions still remain very general inferring the definition of a test from its relationship to programs and specifications.

By themselves, these notions of what constitutes a test case are ambiguous, incomplete, and not general enough. They are ambiguous in that representing a test case in terms of inputs and outputs can be done in many ways. For example, it is not clear whether a composite input should be treated as one or many inputs, or how the expected outputs should be compared against the actual program outputs. They are incomplete in that they do not include critical aspects of a test case like the setup of the context for the SUT to be executed or the fact that multiple inputs may need to be provided at different points during the execution. They are not general enough in that they cannot capture many common types of potential tests like those for non-functional tests that may not rely on an expected output but rather on a performance measure.

The ambiguity and incompleteness of the notations in the literature for what a test case is may partially result from the diversity in the representation of the tests. The representation chosen to specify a test case depends on the several factors, including but not limited to characteristics of the SUT, availability of testing tools, and capabilities of the testing organization. In Section 2.3, we talked about various test representations in the literature, e.g, source code in xUnit frameworks, QTP test scripts, SQL statement, abstract path constraints. For the purposes of comparing and leveraging tests' diverse strengths, however, these representations are not sufficient.

We argue that it is possible to transform test cases into a common representation based on a small number of core components, regardless of how a test case is represented initially. Rather than centering on the inputs and outputs of a test case, our definition of a test (T), is based on three types of interactions between the test case and the SUT: 1) Invoke actions , 2) Write actions, and 3) Decision actions. These three sets of core actions can be sequenced to effectively describe a rich set of interactions between a test and its corresponding SUT, and cover a large spectrum of test types. *Invoke* actions are used to represent the functions invoked to setup the SUT context and the calls to the SUT. *Write* actions represent an update to the SUT state by the test case. *Decision* actions represent a set of comparisons between actual and expected values to judge if the test has passed or failed. This approach is capable of not only capturing the traditional notion of a test case, encoding the inputs and expected outputs of a test case as actions, but it also supports an expanded view of a test case by supporting a more detailed specification of a test case, and by encoding the relative order of the actions in a test case.

$$
\begin{array}{rcl}
T & ::= & t_{ID}\ A\ I_A\ A \\
A & ::= & (I_A \ \mid\ W_A \ \mid\ D_A)^* \\
I_A & ::= & \mathbf{funcSig} \\
W_A & ::= & \mathbf{location}\ \text{“}\leftarrow\text{”}\ \mathbf{vWrite} \\
D_A & ::= & (f_{CMP}(\mathbf{vExpected}, \mathbf{vActual}))^+ \\
\mathbf{funcSig} & ::= & \mathit{fName}\ \ \text{“(”}\ \mathit{fArgTypeList}\ \text{“)”}\ \text{“:”}\ \ \mathit{fRetType} \\
\mathbf{location} & ::= & \mathit{varName} \mid \mathit{freshName} \\
\mathbf{vWrite} & ::= & \mathit{expr} \mid \mathit{constraint} \\
\mathbf{vExpected} & ::= & \mathit{expr} \\
\mathbf{vActual} & ::= & \mathit{expr}
\end{array}
$$

Figure 3.1: TCL syntax

3.1.1 TCL Syntax

Given the procedural nature of many software test cases, we define each test as a sequence of test *actions*. These actions represent updates to the program state, calls to functions in the system under test, and queries of the resulting state to render a validation judge. Figure 3.1 gives the TCL syntax for encoding test cases; literals are enclosed in quotation marks, regular expression notation is used in productions, and named non-terminals are included to enhance the readability of the syntax.

A test, T, is composed of a unique identifier (t_{ID}) and a sequence of actions, A, where at least one action is an invoke action (I_A) representing a call to the method (function) under test. A call to the method under test may occur anywhere in the sequence of actions. An *invoke action* is specified by a function signature defining the name of the function invoked by the test (fName), a list of the function's argument types ($\mathit{fArgTypeList}$), and the function return type ($\mathit{fRetType}$). For example, Figure 3.2 shows a Randoop test for Binary Search Tree, which we used in Section 1.1. An *invoke action* representing the first call to method `insert` in `Test1()` of is specified with the following function signature: `insert(Node, int):void`.

```
public void Test1() throws Throwable{
  Node rootnode = new Node(0);
  insert(rootnode, -1);
}
```

Figure 3.2: Test example for Invoke Action

A *write action* (W_A) represents the assignment of a value ($vWrite$) to a location in memory ($location$). A memory location is specified using the name of a locally or globally defined program variable ($varName$), or a *freshName* when the location cannot be specified by a program variable, i.e., a function argument or the function return value. A fresh variable name can be any string identifier, but must be unique across the set of write actions associated with the same invoke action. For example, consider again the first call to method `insert` in `Test1()` of Figure 3.2. This call to `insert` involves two *write actions*, one for each argument. The first *write action*:

```
insert_arg_0 <-- rootnode
```

specifies the value written to the first argument in the call to `insert` using a fresh name, *insert_arg_0*, and the assigned value, $rootnode$.

The value written ($vWrite$) to a location can be represented as an $expr$ or a $constraint$. An $expr$ is an unspecified typed value defined over constants and variables in the test. A $constraint$ is specified as a conjunction of boolean expressions over constants and variables in the test. For example, Figure 3.3 shows part of the abstract test generated by SPF for Binary Search Tree. The test of `insert` represented by `PC` in Figure 3.3 specifies a constraint over the two arguments of `insert` and, thus, there are two write actions:

```
1) node <-- node.value == 0 &&
node.left == null && node.right == null &&
value > node.value
```

2) `value <-- value > node.value`

These write actions specify the `node` and `value` arguments to `insert` as specified by the constraints generated by Symbolic PathFinder. In Section 3.3 we explain how this encoding of the write actions, using constraints, enables *abstract* test cases to be included in the framework.

```
PC:  node.value == 0 &&
       node.left = null &&
       node.right = null &&
       value > node.value
```

Figure 3.3: Test example for Write Action

A *decision action* (D_A) is a sequence of one or more comparison functions. They are used to represent the comparison of an expected value ($vExpected$), i.e., oracle, with an actual value ($vActual$) to determine the results of the test. The comparison function, $f_{CMP} : Expr \times Expr \rightarrow Boolean$, specifies the comparison operation between the actual and expected values. The comparison operator can be a predefined relational operator, e.g., $\leq$, $\neq$, or a user-specified comparison function. Multiple comparison functions within a single decision action are interpreted as a disjunction.

Sequences of decision actions can be used to encode rich oracles. For example, a logical formula in conjunctive normal form can be expressed as a sequence of decision actions where each encodes a conjunct.

3.2 Transforming Test Cases To the Framework

We demonstrate in this section how the transformation happens from a test case to the common representation in our framework. Given the diversity in test cases, the results of this process will vary considerably in the number and types of actions that are created to represent a single test case. We will illustrate the applicability of TCL (Test Case Language) through a series of examples with different types of tests from concrete tests to abstract tests, and functional tests to non-functional tests.

3.2.1 Manual Tests to TCL

We will illustrate in this section how the transformation to TCL is done for a manual test case testing the soda vending machine.

A vending machine takes in money, a choice of soda from the customer, and outputs customer's choice of soda and any change left after the soda. Imagine in the vending machine we are testing, each bottle of soda costs 1.25 dollars. It has two buttons, one to dispense soda and the other one to dispense the change. A test case to test soda vending machine may include the following steps: (1) a customer inserts 2 dollars, picks the choice of Mountain Dew, (2) when the customer pushes the button to dispense soda, the vending machine should deliver a bottle of Mountain Dew, (3) when the customer pushes the button to dispense the change, the vending machine should give the customer back 0.75 dollars. Figure 3.4 is a more abstract notation of the test case. There are two inputs, two button pushes to the vending machine and the vending machine has one output each time the button is pushed.

The process for transforming a test case to the common test representation is primarily a process of mapping elements in the original representation to the actions specified by the Test Case Language. The inputs are the written values to the memory locations to set up

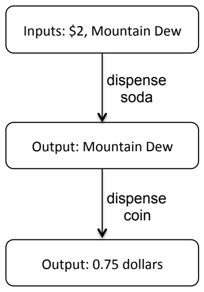


Figure 3.4: Soda Vending Machine Test Case

the program state. A button pushing is a call to the SUT to perform specific functions. The output specified in the test case is the expected value or oracle that the vending machine should deliver. The actual output from the vending machine is the actual value to be checked on. Depending on the implementation of the soda vending machine. The arguments and calls to SUT can be mapped differently. Table 3.1 shows the decomposed test case represented in steps of actions for the testing scenario described above. The column “actionNumber” is used to help connect the the TCL representation in Table 3.2. Table 3.2 is the TCL representation of the manual test for the soda vending machine. Since each type of action contains different components, the table representing TCL is composed of 3 parts: (a) Invoke Actions (b) Write Actions and (c) Decision Actions.

Table 3.1: Decomposed Soda Vending Machine Test Case

Action Type	**Description**	**actionNumber**
W-Action	writes 2 dollars for the money received	0
W-Action	writes “Mountain Dew” for the soda choice	1
I-Action	push button to get soda	2
W-Action	write the soda output to a memory location for the subsequent D-Action	3
D-Action	check if the returned soda is “Mountain Dew”	4
I-Action	push button to get coin change for the subsequent D-Action	5
W-Action	write the output change to a memory location	6
D-Action	check if the coin change is 0.75 dollars	7

Now we will go step by step to see how the transformation process happens from Table 3.1 to the TCL representation in Table 3.2. The inputs of 2 dollars and "Mountain Dew" are W-Action in Table 3.2, as input arguments for the vending machine with *actionNumber=0,1*. Pushing the button to get the soda is mapped in Table 3.2 to an invoke action for the call the the SUT (*actionNumber=2*). A W-Action (*actionNumber=3*) is added to capture the output of the call to be used as the actual value in the subsequent assertion. The *value-Expression* for the write action is the value that is written. In Table 3.2, the write action (*actionNumber=3*) has a *value-Expression* of *Soda_1*. *Soda* refers to the type of the returned output and 1 refers to the index of such type in the current test case. *Soda_1* means this is the first value written as type *Soda*. The D-Action (*actionNumber=4*) in Table 3.2 is the decision action performing the comparison between the expected value specified in the test case and the actual value output from the call to the SUT. The transformation for I-Action (*actionNumber=5*), W-Action (*actionNumber=6*) and D-Action (*actionNumber=7*) is similar to that of I-Action (*actionNumber=2*), W-Action (*actionNumber=3*) and D-Action (*actionNumber=4*).

Table 3.2: Soda Vending Machine Test Case in TCL

actionNumber	**funcSig**
2	Soda getSoda()
5	double getChange()

(a)Invoke Actions

actionNumber	**location**	**vWrite**
0	VendingMachine_money_input	2.0
1	VendingMachine_sodachoice_input	"Mountain Dew"
3	getSoda_output	Soda_1
6	getChange_output	Double_1

(b)Write Actions

actionNumber	**vExpected**	f_{CMP}	**vActual**
4	"Mountain Dew"	$=$	Soda_1
7	0.75	$=$	Double_1

(c)Compare Actions

From Section 3.2.2 through Section 3.2.5, we will be using the same table structure to demonstrate the TCL representation for other types of tests.

3.2.2 JUnit Tests in TCL

JUnit test cases are rather standardized, yet, they can vary depending on which tool has generated them or if they are written manually. For example, the Randoop test generation tool mostly generates tests in straight-line code with a limited number of try/catch blocks, but manually designed JUnit tests sometimes utilized complex class hierarchies to make the test suite more concise. In all JUnit tests, the order of the statements indicates the sequence of actions. Each assignment or argument setup is a write action to a memory location. The usage of calls to assertions such as `assertEquals` signals a decision action. Moreover, we can identify (because we have JUnit API information for these function calls) which value is expected and which is the actual value in the assertion.

Figure 3.5 is a hand-coded JUnit test case for the ACCoRD framework [1]. ACCoRD is used at NASA to implement the state-based conflict detection and resolution algorithms. To help understand the transformation from the test case to TCL, the intermediate representation in Table 3.3 shows the relationship between the line of code and the actions in TCL. Table 3.4 shows the test case in TCL.

```java
public void testAddLL() {
    Plan fp = new Plan();
    fp.addLL( 1.0, -1.0, 5000.0, 10.0);
    assertEquals(1, fp.size());
}
```

Figure 3.5: Example to Show How TCL Represents the JUnit Tests

Table 3.3: Decomposed JUnit Test Case

Line #	**Action Type**	**Description**	**actionNumber**
1	I-Action	invoke constructor to create Plan object	0
1	W-Action	capture the memory location of the Plan object created	1
1	W-Action	assign the Plan object to its lhs variable *fp*	2
2	W-Action	write the 1*st* argument for addLL	3
2	W-Action	write the 2*nd* argument for addLL	4
2	W-Action	write the 3*rd* argument for addLL	5
2	W-Action	write the 4*th* argument for addLL	6
2	I-Action	invoke the method *addLL(double,double,double,double)*	7
3	W-Action	write the 1*st* argument for *assertEquals*	8
3	I-Action	invoke method *size()* on *fp*	9
3	W-Action	capture the memory location of returned int value	10
3	W-Action	write the returned int value as the *2nd* argument for *assertEquals*	11
3	D-Action	check if the returned int value equals to the expected value 1	12

Table 3.4: JUnit Test Case in TCL

actionNumber	**funcSig**
0	Plan Plan()
7	void Plan.addLL(double,double,double,double)
9	int Plan.size()

(a)Invoke Actions

actionNumber	**location**	**vWrite**
1	Plan_return_value	PlanObject1
2	fp	Plan_return_value
3	fp.addLL_arg0	1.0
4	fp.addLL_arg1	-1.0
5	fp.addLL_arg2	5000.0
6	fp.addLL_arg3	10.0
8	assertEquals_arg0	1
10	fp.size_return_value	intValue1
11	assertEquals_arg1	fp.size_return_value

(b)Write Actions

actionNumber	**vExpected**	f_{CMP}	**vActual**
12	1	$=$	fp.size_return_value

(c)Compare Actions

3.2.3 Differential Unit Test in TCL

Figure 3.6 (pre-state) and Figure 3.7 (post-state) represents the original test case that is generated by the Carving tool [23]. Table 3.5 is the intermediate representation showing the relationship between the line in XML file and the actions in TCL, and Table 3.6 shows the test case in TCL. The target method of this test is *double Velocity.track()*. The pre-state in Figure 3.6 captures the state of the receiver object used to call the method under test. There are no other arguments because the target method *double Velocity.track()* does not take in any arguments. The post-state in Figure 3.7 captures the receiver object state after the method invocation and the return value.

We discuss the implementation of the transformation process from DUT to TCL in Section 3.3.

```
1:   <object-array>
2:     <gov.nasa.larcfm.Util.Velocity>
3:       <__INSTANCE__>
4:         <x>22225.128251356447</x>
5:         <y>-22221.74325667393</y>
6:         <z>-121.92</z>
7:       </__INSTANCE__>
8:       <__STATICS__/>
9:     </gov.nasa.larcfm.Util.Velocity>
10:    <object-array/>
11:  </object-array>
```

Figure 3.6: Example to Show How TCL Represents the DUTs (pre-state)

```
1:   <object−array>
2:     <gov.nasa.larcfm.Util.Velocity>
3:       <____INSTANCE____>
4:         <x>22225.128251356447</x>
5:         <y>−22221.74325667393</y>
6:         <z>−121.92</z>
7:       </____INSTANCE____>
8:       <____STATICS____/>
9:     </gov.nasa.larcfm.Util.Velocity>
10:    <double>2.3561183319710546</double>
11:  </object−array>
```

Figure 3.7: Example to Show How TCL Represents the DUTs (post-state)

Table 3.5: Decomposed DUT

Line #	**Action Type**	**Description**	**actionNumber**
pre-state 4	W-Action	write the $1st$ argument for $Velocity$ constructor	0
pre-state 5	W-Action	write the $2nd$ argument for $Velocity$ constructor	1
pre-state 6	W-Action	write the $3rd$ argument for $Velocity$ constructor	2
pre-state 2,9	I-Action	invoke constructor to create $Velocity$ object	3
pre-state 2,9	W-Action	capture the memory location of the $Velocity$ object	4
pre-state 2,9	W-Action	assign the returned $Velocity$ object to a receiver for subsequent method invocation	5
method invocation	I-Action	invoke method $trace()$ on $receiverObj$	6
method invocation	W-Action	capture the memory location of returned double value	7
method invocation	W-Action	assign the double return value for its use in the subsequent decision action	8
post-state 4	D-Action	check attribute x in the receiver object after method invocation	9
post-state 5	D-Action	check attribute y in the receiver object after method invocation	10
post-state 6	D-Action	check attribute z in the receiver object after method invocation 1	11
post-state 10	D-Action	check if the returned double value equals to the expected value 2.3561183319710546	12

Table 3.6: Differential Unit Tests in TCL

actionNumber	**funcSig**
3	Velocity Velocity(double,double,double)
6	double Velocity.trace()

(a)Invoke Actions

actionNumber	**location**	**vWrite**
0	Velocity_arg_0	22225.128251356447
1	Velocity_arg_1	-22221.74325667393
2	Velocity_arg_2	-121.92
4	Velocity_return_value	VelocityObject1
5	receiverObj	Velocity_return_value
7	receiverObj.trace_return_value	doubleObject1
8	returnValue	receiverObj.trace_return_value

(b)Write Actions

actionNumber	**vExpected**	f_{CMP}	**vActual**
9	22225.128251356447	$=$	receiverObj.x
10	-22221.74325667393	$=$	receiverObj.y
11	-121.92	$=$	receiverObj.z
12	2.3561183319710546	$=$	returnValue

(c)Compare Actions

3.2.4 Abstract JPF Tests in TCL

Figure 3.8 is the method under test. Figure 3.9 is the original path constraints generated by JPF for the method under test, Table 3.7 is the intermediate representation of the test case and Table 3.8 shows the abstract test case in TCL.

First, we map the symbols in the path constraints to the arguments of the method under test. For value-expression in the write actions for the path constraints, only the clauses that have the write-location involved are included in the expression. For example, for the value-expression of "Util.within_epsilon_arg_0", only clauses that involve "arg_0" are included.

```
public static boolean within_epsilon(double a, double b, double epsilon) {
        return Math.abs(a–b) < epsilon;
}
```

Figure 3.8: Example to Show How TCL Represents Abstract Test Cases - source code

```
(CONST_0.0 − (a_4_SYMREAL[−10000.0] − b_5_SYMREAL[−9999.0])) <
epsilon_6_SYMREAL[2.0] && (a_4_SYMREAL[−10000.0] − b_5_SYMREAL[−9999.0]) < CONST_0.0
```

Figure 3.9: Example to Show How TCL Represents Abstract Test Cases - path constraint

Table 3.7: Decomposed Abstract JPF Test Case

Action Type	**Description**	**actionNumber**
W-Action	write the constraint for the $1st$ argument	0
W-Action	write the constraint for the $2nd$ argument	1
W-Action	write the constraint for the $3rd$ argument	2
I-Action	invoke the method under test $within_epsilon(double, double, double)$	3

Table 3.8: JPF Abstract Tests in TCL

actionNumber	**funcSig**
3	Util within_epsilon(double,double,double)

(a)Invoke Actions

actionNumber	**location**	**vWrite**
0	Util.within_epsilon_arg_0	(CONST_0.0 - (arg_0 - arg_1)) <arg_2 && (arg_0 - arg_1) <CONST_0.0
1	Util.within_epsilon_arg_1	(CONST_0.0 - (arg_0 - arg_1)) <arg_2 && (arg_0 - arg_1) <CONST_0.0
2	Util.within_epsilon_arg_2	(CONST_0.0 - (arg_0 - arg_1)) <arg_2

(b)Write Actions

There are no explicit decision actions in this test.

(c)Compare Actions

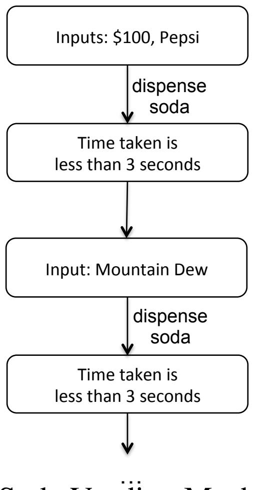


Figure 3.10: Soda Vending Machine Load Test

3.2.5 Load Tests in TCL

We explore in this section the transformation of a load test into TCL. We use a load test for testing the same soda vending machine we discussed in Section 3.2.1. Imagine in a testing scenario where we insert into the vending machine a 100-dollar bill, make continuous choices of soda as quickly as we can and use a stop watch to record the time it takes to get the soda each time. The specification indicates that it should take less than 3 seconds to get one bottle of soda, and this serves as the oracle of the load test. The soda costs 1.25 dollars, so we can get 80 bottles, but we abbreviate the process in the transformation to demonstrate only two. The first choice made by customer is “Pepsi” and the second choice is “Moutain Dew”. Also, in a pure load test, we do not check the functional correctness of the SUT. In other words, we do not check whether the returned soda is the choice made by the customer. Note that the load tests can be integrated with functional tests to check both the correctness of the output choice of soda and the time taken to get the soda.

Figure 3.10 demonstrates the process for the specific load test scenario of the soda machine. Table 3.9 represents the decomposed test case in detailed steps. Table 3.10

delivers the corresponding TCL representation of the test case. The only difference in this load test from the manual functional test discussed in Section 3.2.1 is the type of oracles being checked.

Table 3.9: Decomposed Soda Vending Machine Load Test

Action Type	**Description**	**actionNumber**
W-Action	writes 100 dollars for the money received	0
W-Action	writes “Pepsi” for the soda choice	1
I-Action	push button to get soda	2
W-Action	write the time taken to a memory location	3
D-Action	check if the time taken is less than 3 seconds	4
W-Action	writes “Mountain Dew” for the soda choice	5
I-Action	push button to get soda	6
W-Action	write the time taken to a memory location	7
D-Action	check if the time taken is less than 3 seconds	8

Table 3.10: Soda Vending Machine Load Test in TCL

actionNumber	**funcSig**
2	Soda getSoda()
6	Soda getSoda()

(a)Invoke Actions

actionNumber	**location**	**vWrite**
0	VendingMachine_money_input	100.0
1	VendingMachine_sodaChoice_input	“Pepsi”
3	timetaken_output	Double_1
5	VendingMachine_sodaChoice_input	“Mountain Dew”
7	timetaken_output	Double_2

(b)Write Actions

actionNumber	**vExpected**	f_{CMP}	**vActual**
4	3 seconds	$\geq$	Double_1
8	3 seconds	$\geq$	Double_2

(c)Compare Actions

3.3 Implementation

To evaluate our technique, we implemented a tool to automate the transformation of test cases into the common test representation presented in Section 3.1. Our tool is capable of transforming test cases specified as Java source code, e.g., JUnit tests, and test cases specified in the custom format used by the test carving tool described in [23] for generating differential unit tests (DUTs). However, other implementations could be easily added.

Transforming Java Source Code When the input to the transformation process is Java source code, the transformation is performed in three phases. The first phase uses the Abstract Syntax Tree (AST) for the test case (generated by the Java compiler) to map the elements in the AST to the actions in the Test Case Grammar. The order of the statements in the source code is used to sequence the actions in the transformed version of the test case.

The second phase of the transformation process is to propagate the *setup* in the test class if it exists. In the JUnit framework, the *@Before* annotation or *setup* methods can be incorporated to have methods executed before each test is run. In the setup propagation phase, we add the actions for the setup methods before the actions in each of the tests included in the same class such that each test can be treated as an individual test.

The third phase is the dereference phase. In this phase, we propagate the variable values through the actions. In a statement, when there is use of a variable name referring to a value in the preceding statements, we propagate the actual value that was assigned to the variable from the preceding statement. This does not change the semantics of the tests, but it is to help us understand the statements better in the sense that we know what the actual values are in the memory locations related to the statement. For example, for an assertion *assertEquals(1, var1)*, before dereference we know the test is comparing *var1* against the expected value of *1*. However, after dereference, we will know the exact memory location that the test wants to compare which can be the return value of a method call. Another case when this is desirable is when there is a reference in the test to the field by its name. In order to analyze the test separately, it is desirable to know the values that the field name holds.

```
public void test7() throws Throwable {
  // The following exception was thrown during execution.
  // This behavior will recorded for regression testing.
  try {
    double var3 = gov.nasa.larcfm.Util.Units.convert(``hi!'', ``hi!'', (-0.1493072d));
    fail(``Expected exception of type gov.nasa.larcfm.Util.Units.UnitException'');
  } catch (gov.nasa.larcfm.Util.Units.UnitException e) {
    // Expected exception.
  }
}
```

Figure 3.11: Randoop Test with try/catch Block

In the current implementation of the framework, we handle straight-line code with limited support for branching. There are two specific cases when we handle branching. The first one is when the test generator is Randoop, we handle the try/catch block. Randoop has a special template that we follow during implementation. Figure 3.11 is an example of one Randoop test, and this is the only case when Randoop has branching in its generated tests. We process the first statement normally, and then process the second statement in the block as a decision action, ignoring the catch block in the test since there are no statements in that block.

The second one is when the tests are translated from JML specification, we handle *if* branches. Figure 3.12 is one example of the tests. It also has a template that we can follow to perform the transformation. All the if-conditions in the tests represent the precondition of the method under test and serve as a guard for the assertions to happen. Thus, during our transformation, we process the first statement normally and then the condition in the predicate serves as a condition of the decision actions which represent the assertions

```
public void testAdd() {
  Rectangular r1 = new Rectangular(−10.0, 75.2);
  Polar p2 = new Polar(7.5, StrictMath.PI / 3);
  if(p2!=null){
    Complex c3 = r1.add(p2);
    assertNotNull(c3);
    assertEquals(r1.realPart() + p2.realPart(), c3.realPart(), 0.005);
    assertEquals(r1.imaginaryPart() + p2.imaginaryPart(), c3.imaginaryPart(),0.005);
   }
}
```

Figure 3.12: Hand-code tests translated from JML with if branch

statement inside the block. In the evaluation in Chapter 5, we incorporated hand-coded tests translated from JML specification.

Transforming Abstract Test Specification To transform abstract test cases into the TCL, we create one *write* action for each program variable, e.g., method argument or global variable, specified in the path condition. As we discussed in Chapter 3, the *value written* (*vWrite*) is represented by the conjunction of constraints over the program variable specified in *location*. An *invoke* action is created to represent the call to the unit under test. We implemented this transformation using a custom application to parse and transform the path conditions generated by Symbolic PathFinder into the test case language.

Transforming Differential Unit Tests in XML Files The input to the transformation process when the test cases have been generated by the test carving tool is a set of XML files, each of which contains a unit-level test case. Each test case specification is composed of a pre-state, the name of the unit under test, and the post-state. The pre-state represents the program state before the method under test is invoked and the post-state represents the program state after the method invocation. For carved test cases, the transformation maps the elements in the XML structure to the actions in the Test Case Language. The sequence of actions is dependent on the element order in the XML files. In the pre-state of the DUT, it first includes the the receiver object, and then comes the list of arguments needed to invoke the method under test. During transformation, we first create the receiver object according to the values captured in the pre-state. Then we add a write action for each of the argument needed for the method invocation and finally add one invoke action to represent the method call. The elements in the post-state are treated as the "oracles" for the test. The post-state is composed of two parts - receiver object and the return value if there is one. We add one assertion for each attribute of the receiver object and finally one for the return value.

We note that there are a couple of potential challenges interpreting DUTs. First, from the pre-state, we know the exact state of the receiver object, the values being assigned to its attributes. However, we may not be able to set up the object state without knowing what method-call sequences to use. Second, with the post-state, we would like to check the state of the object after the method invocation, but we do not know for sure the visibility of the attributes of the object in the corresponding class. It is possible to solve the second problem with access to the SUT. For example, with access to the SUT, we can either know the visibility of the attributes or search for *getter* methods which return the desired attributes. However, the first problem of setting up the receiver object state remains challenging. Our current implementation make approximations in order to represent the rich object state information contained in the DUTs. We assume that there is a constructor in the class that does not take in any arguments and the fields in the class are public. To set up the object state in the pre-state, we first create an object with a constructor of the class which does not take in any argument, and then we add a write action for each of the attribute of the object. To represent the post-state, we create decision actions with direct access to the fields of the class.

Storing Tests' TCL Representation Each test is stored as an *ActionSummary*, which is composed of a list of *invoke action*, *write action*, *decision action*, and some additional information like *testName*, *className*, *sourceFile* and *Id*.

In *invoke action*, besides the *funcSig* described in the grammar, it may also include *receiver object* on which the method was invoked and a *related write action* that captures the corresponding write action that the return value of this invoke action writes to. If the method being invoked is a constructor, the *receiver object* is null, and if there is no return value like a setter method, the *related write action* is empty.

In *write action*, besides the *location* and *vWrite* described in the grammar, it may also have an *object* if the location is an attribute of one particular object. We also infer the type information from the test and store the *type* of the *location* being written.

In *decision action*, it has an *vExpected* representing the oracle of the decision action, a *vActual* representing the actual value captured in the test, and a f_{CMP} denoting the relationship between the two values. f_{CMP} may also have a *message* field to include the message that comes with the assertion and a *precision* value for the imprecision tolerance between *vExpected* and *vActual*. It may have a *condition* in some cases which represents the pre-condition for this decision action to take place. In other words, the decision action is performed only when that condition is satisfied.

We also created a database to store the *ActionSummary* of tests. This is to facilitate some light-weight analysis. For example, instead of writing a complex analysis in Java language, you can write simple SQL queries to find out how many decision actions are utilized in one test, one test class or one test suite, or how many invocations happened for one target method, etc.

3.4 Assessing the TCL

In the discussions above, we illustrate the expressiveness of our test case language and its flexibility to encode concrete and abstract test cases generated using both automated and manual techniques. In Section 5 we implement and evaluate a set of test advice functions which use the tests in the framework to generate amplified test suites. The evaluation of manual test cases includes hand-coded test cases from the ACCoRD framework [1]. Of the 180 manual tests available, we were able to transform 157 into the framework. Of the 23 tests our transformation could not handle, 10 included conditional control flow not supported by the transformer, and the other 13 tests were effectively composite test cases that called other test methods to perform a series of tests. To include these test cases, we could have in-lined the test methods and processed the composite test as a single test case. Our transformation processes were able to process all of the more than 500-thousand auto-generated test cases (from Randoop and Symbolic PathFinder) – refer to Chapter 5 for the actual numbers of auto-generated test cases.

One of the limitations of our transformation algorithms is related to the write actions that are generated – any updates to the state that occur outside the test will not be captured. This is because we transform a static representation of the test, and we do not access the SUT (in a static or dynamic manner) so our only input is the source code for the test. This was a conscious decision to avoid tying the test to a particular version of the SUT; however, the trade-off is that our tests may miss the write actions resulting from calls to other methods.

Finally, it is worth noting that the tests transformed in this work were all *functional* tests. We conjecture, however, that our test case language will also support non-functional, e.g., load, tests. One of the main differences will be in the specification of the *decision* actions, which for non-functional tests, will involve comparisons of resources used with specified threshold values.

Chapter 4

Test Advice

In Chapter 3, we discussed the common representation of tests, the implementation of the framework, and the different types of tests that the framework supports. In this chapter, we will elaborate on how we make effective use of the test data collected in the framework to provide a family of advice. Figure 4.1 provides a general overview of the approach we take. We assume that a user of the framework will define a testing context, $\mathbb{C}$, which identifies, for example, a model, a specification, or more commonly an SUT. The overall operation of the framework exploits this context.

Tests are *transformed* into a common test encoding, $\mathbb{F}$, using standard language processing techniques – parsing, tree transformation, and code generation. Transformed tests are collected in a *test case language* (TCL) repository for future use. Tests in TCL can easily be converted back to different test representations, i.e., we can map $\mathbb{F}$ to a set of tests, $\mathbb{T}$.

The structure of TCL permits the easy *extract*ion of different types of information that is related to the testing context; what we refer to as *test advice*, $\mathbb{A}$. For instance, sequences of test operations that define portions of an SUT's input state (potential input advice), and tests of SUT output values (potential oracle advice) are made explicit in the syntax of TCL.

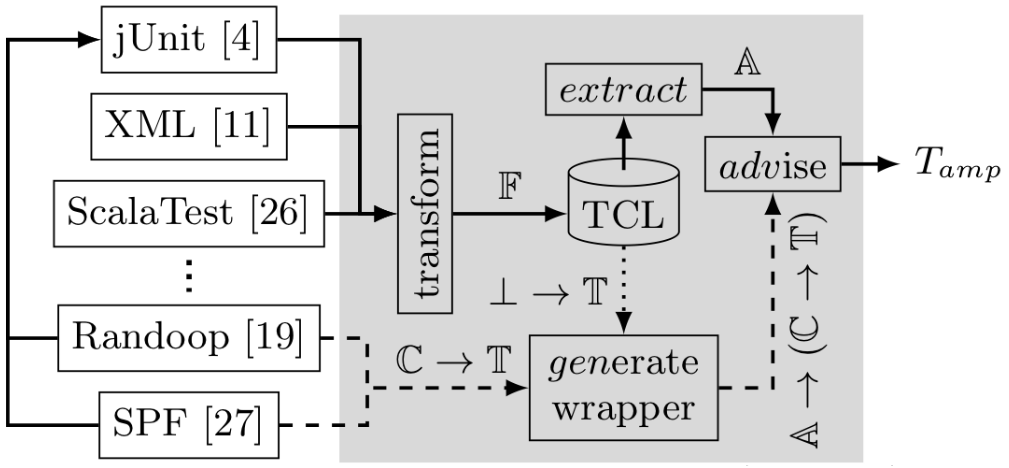


Figure 4.1: Test Advising Approach

Test advice can be exploited by existing tests and test case generation tools alike. This is depicted in two ways through the dotted and dashed flows through the *generate wrapper* component. (1) The repository of TCL tests can be queried for a set of tests that are relevant to the testing context – the dotted flow. For uniformity with the second case, the test set is represented as a constant (nullary) function, $\perp \to \mathbb{T}$. (2) We view test generators as functions, $\mathbb{C} \to \mathbb{T}$, that take a testing context and produce a set of tests – the dashed flow.

Wrapper generation involves exposing an interface to these functions that permits additional information to be leveraged during their application. In the first case this is fully automatic, but for existing monolithic test generators this must be achieved through either existing APIs, as in the case of Randoop's literal constant pool, by providing the tool an enhanced test harness, as in our usage of SPF, or, in future work, by modifying generator implementations. Wrapping results in a function, $\mathbb{A} \to (\mathbb{C} \to \mathbb{T})$, enables extracted advice to be applied to customize the test generation process to produce amplified tests, T_{amp}. We call this step, and the overall process described in Figure 4.1, *test advising*.

In the rest of the Chapter, we start with the formal definition of a set of functions for extracting and generating testing advice. We then describe three families of advice functions that leverage the features of the TCL to generate testing advice targeting a particular test component, i.e., type of action, in the TCL.

4.1 Test Advice Functions

Testing advice can be extracted from a wide range of frameworks, i.e., sources. We assume the *encoding* of the data in the framework will vary depending on factors such as the tool or technique used to generate the data, and on the type of the data, e.g., specification, source code. We also assume the encoding of the advice itself may vary. Let $\mathbb{F}$ be a type representing the possible test framework encodings. For the testing advice specified in this work, we use the test case language encoding specified in Chapter 3 as a concrete instance of $\mathbb{F}$. Let $\mathbb{A}$ be the type representing the advice encodings. Given these type definitions, we define a function for extracting advice from a set of tests:

Definition 4.1.1 (Advice Extraction Function) *An advice extraction function,* $extract\colon \mathbb{F} \to \mathbb{A}$*, computes advice from a set of test framework encodings.*

The inputs to a software test generation technique can vary widely, ranging from the domain knowledge residing in the test developer's head, to formal specifications of expected behaviors, to the source code for the SUT. Let $\mathbb{C}$ be a type representing such inputs – we term this *test generation context*. Note that our technique is not concerned with the specifics of the test generation process or context. It is, however, necessary to establish a shared context that is used to coordinate advice extraction and test generation, i.e., the extracted advice should be appropriate for the test generation context.

The output of a test generation technique is a *test suite* consisting of one or more test cases. Let $\mathbb{T}$ be a type representing a set of test cases. Given these type definitions, we now define a function for generating a test suite:

Definition 4.1.2 (Test Generation Function.) *A test generation function, $gen\colon \mathbb{C} \rightarrow \mathbb{T}$, produces a set of test cases, i.e., test suite, based on the given testing context.*

We note that a fixed test suite is defined by a nullary instance of gen whose domain is empty, i.e., $\mathbb{C} = \perp$.

In order to incorporate advice into the test generation process we must extend gen so that it can accept and apply advice. We do this by defining a higher-order *test advice function* which effectively wraps instances of gen.

Definition 4.1.3 (Test Advice Function) *A test advice function, $adv\colon \mathbb{A} \rightarrow (\mathbb{C} \rightarrow \mathbb{T})$, uses extracted test advice to produce an enhanced test generation function which can, in turn, produce a test suite.*

Instances of this function will be created and applied to generate an enhanced test suite, $T_{amp} = adv(extract(F)(C))$, exploiting advice from a given set of tests encoded in the framework and a test generation context. Note that evaluating adv produces a customized test generation function which is then applied, with an appropriate testing context, to produce amplified tests.

4.2 A Family of Advice Functions

A test, in general, needs $inputs$ which are used to set and modify the program states, $calls$ which are used to perform the functionality of the system, and $oracles$ which are needed to check the correctness of the results of the test. In the context of Object-Oriented languages, three fundamental pieces are desirable to compose a “cost-effective” test: (1) input values used to invoke methods, (2) method sequences utilized to set up the program state to explore, (3) the oracle checks that help decide whether the test passes or fails. In this section, we define a family of concrete test advice functions that leverage the test case

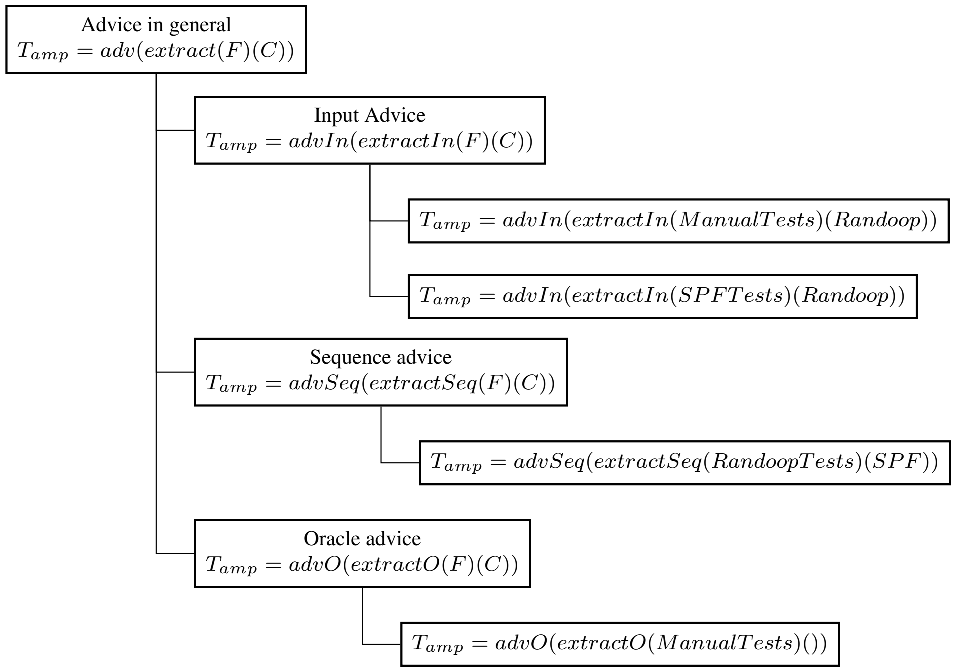

Advice in general
$T_{amp} = adv(extract(F)(C))$
Input Advice
$T_{amp} = advIn(extractIn(F)(C))$
$T_{amp} = advIn(extractIn(ManualTests)(Randoop))$
$T_{amp} = advIn(extractIn(SPFTests)(Randoop))$
Sequence advice
$T_{amp} = advSeq(extractSeq(F)(C))$
$T_{amp} = advSeq(extractSeq(RandoopTests)(SPF))$
Oracle advice
$T_{amp} = advO(extractO(F)(C))$
$T_{amp} = advO(extractO(ManualTests)())$

Figure 4.2: Test Advices based on Test Advising Framework

language presented in Chapter 3. Each function is focused on providing advice to improve either the test inputs, the sequences of function calls, or the test oracle. The simplicity of these functions illustrates the power of the TCL to enable tests to be treated as a source of advice for improving other tests. Figure 4.2 gives an overview of the four pieces of advice we have instantiated and explored in this thesis. In Chapter 5, we evaluate each concrete instantiation of the advice functions.

4.2.1 Advice on Test Input Values

The input values used to invoke methods in tests can have a significant impact on the quality of the tests. Missing some input values may weaken any sophisticated setup of program state and leave certain parts of the SUT untested. On the other hand, having too many input values lying in the same input partition will increase the maintenance and execution cost of the tests. In this section, we address the problem of missing important input values.

4.2.1.1 Motivation

We now present a small example to show the importance of input value selection.

```
1 public void example(int x){
2     if(x < -10){
3         //bug here
4     }else{
5         //everything is fine
6     }
7 }
```

Figure 4.3: Example to demonstrate the importance of input value selection

For the code shown in Figure 4.3, only the tests invoking method *example* with input values less than -10 are capable of revealing the bug. Now consider a random test generation tool that uses a pre-defined pool of input values to choose from. If the pool does not include an integer value that is less than -10, the test suite generated by the random approach will reach the *if* predicate at line 2, but fail to cover the code inside the predicate, and thus fail to reveal the bug. On the other hand, manual tests may include values such as -9, -10, -11. Advising random test generation tool with those values will be of particular help under such circumstances.

4.2.1.2 Definitions

INPUT VALUE In general, an *input value* is the value of the write action assigned to a memory location. Input values are used to set the program state before invoking the functions in SUT. In our framework, a method input value used to invoke the method under test is represented as a W_A(write action). In the context of unit tests for an object-oriented language, our instantiation takes the following form:

Given a write action, $W_A = \textbf{location}$ "$\leftarrow$" $\textbf{vWrite}$, an $input\ value$ is the written-value in a W_A where *location=freshName* denoting *methodName_arg_index*.
In *methodName_arg_index*, *MethodName* is the name of the method invoked, and *index* is the order index in the argument list starting from 0.

SCOPE-VALUES PAIR In general, the input values or variables in a program all have their own scope whether it is a block, a method, a class, or global. In the setting of our framework, we also have scope for values to indicate when and where they can be used. In a scope-values pair (scope, values), SCOPE can have 4 different granularities, namely $method$, $class$, $package$, $program$. VALUES refers to the set of $input\ values$ associated with the scope (note that we use a set since there are no duplicates).For instance, $values$ will be the set of the arguments for a method if the scope is $method$, and value will be the set of arguments used for all methods in a given class if the scope is $class$.

Also, in our framework, the scope of input values is not explicitly saved in terms of the 4 granularities. We save the *(method, values)* information.Inferring the class to which each method belongs, we know which class scope the values have. Also inferring the package to which this class belongs, we know which package scope the values have.

INPUT ADVICE FUNCTION

An input advice function, $advIn$, leverages advice in the test framework to determine the test inputs that may help to improve a test suite.

We define $extractIn$: $\mathbb{F} \rightarrow \mathbb{A}$ which extracts the test input data from the tests stored in the framework to generate advice A_{In}. A natural instantiation of $advIn$ extracts the test inputs from the $vWrite$ component of the *write* actions. For each input i extracted from the framework, a *scope*, s, is inferred to identify where the input is used within the test. The scope can be specified at various levels of precision: $method$, $class$, $package$, or $program$, and is inferred from the context $\mathbb{C}$.

For this instantiation of $advIn$, the advice computed by $extractIn$, $A_{In} = \{(i_1, s_1), (i_2, s_2), ..., (i_n, s_n)\}$, is used to control the input values produced by test generator. In Section 4.2.1.3 and Section 4.2.1.4, we explain the implementation details of two concrete instances of the test input advice function.

4.2.1.3 From Hand-Coded Tests to Randomly Generated Tests

Hand-coded tests encode the programmers' understanding of the system under test into the tests they write. Input values picked for the methods being invoked is one aspect of such knowledge. On the other hand, as the example shown in Figure 4.3, random testing approach may have the problem of lacking certain input values needed to cover desirable statements in the SUT. In this section, we explore the potential benefits that hand-coded tests may bring to randomly generated tests by providing advice on method input values. In this thesis, we use Randoop as our representative of the random test generation tools. More specifically, this input advice function is:

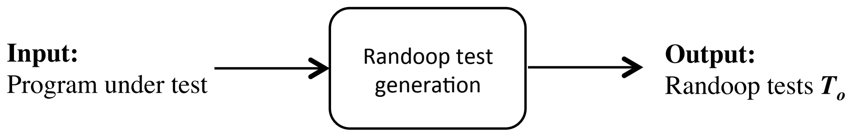
$$extractIn : ManualTests \rightarrow Inputs,$$

$$gen : Randoop \rightarrow RandoopTests,$$

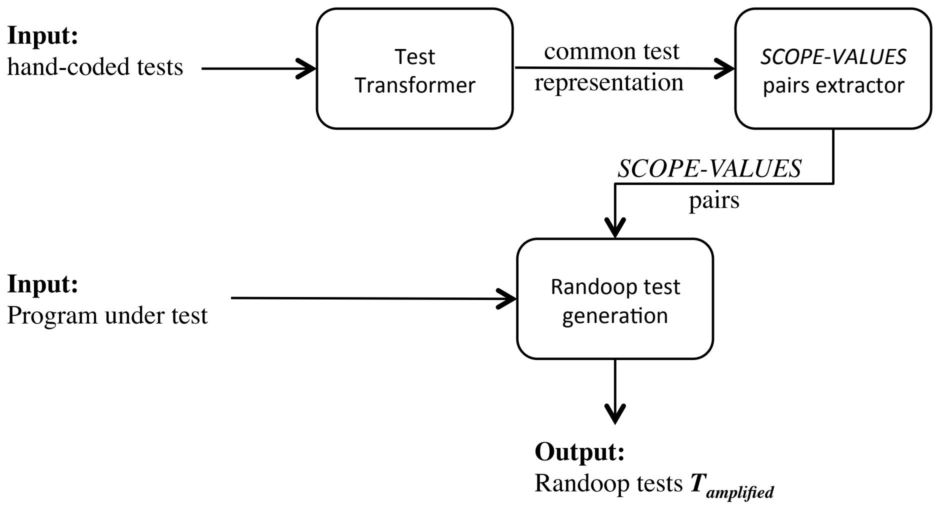
and

$$T_{amp} = advIn((Inputs)(Randoop))$$

The scope for the $Inputs$ advice is computed at the class-level.



(a) without input advice from Hand-coded Tests



(b) with input advice from Hand-coded Tests

Figure 4.4: Process to Generate Randoop Tests with and without input advice from Hand-coded Tests

Implementation The following is the two-step implementation we performed to deploy the test method input advice from hand-coded tests to Randoop tests.

Step 1, given a test suite $ManualTests$ (hand-coded test suite), we collected the set of method input values $Inputs$. We extracted all the $class\text{-}values$ pairs from the hand-coded test suite. We first transformed the hand-coded test suite into the framework representation and filtered the tests to get the subset of write actions whose location is a *freshName* for method argument. This step helps us to get all the $class\text{-}values$ pairs used for method invocations in the original hand-coded test suite.

Step 2, we advise Randoop to use the additional inputs extracted from step 1 for its test generation. Randoop has a pre-defined initial pool of values to use for test generation. Randoop picks from -1, 0, 1, 10, 100 for numeric values, '$\#$', ' ', '4' and 'a' for characters, and "", "hi!" for strings. And any other input values needed will be unavailable for Randoop. With the advice, Randoop chooses from the original pool plus the advice provided by the hand-coded tests. Since Randoop currently does not incorporate any reference values from an external source during test generation process, we further restrict our advice to numeric and string types.

Figure 4.4 summarizes the process we use Randoop to generate tests with and without the input advice function applied. We perform an evaluation of this concrete advice function in Section 5.3.1.

4.2.1.4 From Systematically Generated Tests to Randomly Generated Tests

In this section, we take another approach to the input advice, replacing the values extracted from hand-coded tests in Section 4.2.1.3 with input values generated from systematic test generation approaches.

As discussed in Section 2.2, one approach to systematic testing is Symbolic Execution. This approach executes the method sequences of the SUT using symbolic values for method parameters, and collects path constraints on these parameters along the execution. Finally, it utilizes underlying solvers to solve the constraints collected with symbolic parameters and the solution creates actual concrete test input values for the method parameters. Since these concrete input values are needed by the tests to cover partitions of the SUT's total input space, they provide an excellent source of input value advice. In this section, we evaluate how these carefully chosen input values from systematic testing approaches can benefit random test generation techniques. In our implementation, we use Randoop [53] as representative of the random test generation approaches, and use SPF [55, 56] for the systematic approach.

More specifically, this input advice function is:

$$extractIn : SPFTests \rightarrow Inputs,$$

$$gen : Randoop \rightarrow RandoopTests,$$

and

$$T_{amp} = advIn((Inputs)(Randoop))$$

The scope for the $Inputs$ advice is computed at the class-level.

Implementation Following the test driver example that Visser et al. wrote for the Java containers [74], we wrote test drivers for the artifacts we are experimenting on. In the test driver, we explicitly specified the sequence length and all the methods we need to involve in the test generation process. SPF explores the method sequences by exercising the methods specified in the test driver. We used the SymbolicSequenceListener while running symbolic execution of the SUT. This listener helped to produce the generated test sequences in the JUnit format.

Once we had the systematically generated test suite from SPF, the rest of the implementation for this part is the same as the implementation described in Section 4.2.1.3 and in Figure 4.4. First, we extracted all the $(class\text{-}values\ pairs)$ from SPF test suite. Then we advised Randoop to generated tests based on the advice on these $(class\text{-}values\ pairs)$ extracted from the SPF tests.

4.2.2 Test Method Sequence Advice

A test method sequence consists of a series of method calls in a test to set up the program state. There are usually different interpretations of the method sequence concept. The method sequence we discuss in this section refers to the well-formed method sequence associated with proper data dependences enforced. By this, we mean subsequent statements in the method sequence refer to preceding statements for data to use.

4.2.2.1 Motivation

The method sequence in a test decides what program states the test can reach and consequently what oracle checks can be performed. Galeotti et al. illustrated in [28] one test example for one implementation of BinomialHeap, where a method sequence of length 13 is required to cover part of the source code and to reveal an underlying bug. Sai et al. also experienced similar situation in [82] with Randoop. Randomized test generation struggles to compose a method sequence with specific order and particular arguments. For example, Randoop is not able to generate the method sequence required to make a connection to a database system.

A lot of research exists on generating meaningful method sequences as test input [46, 52, 73, 74, 76]. In Section 4.2.2.3 we will address this challenge with the help of method sequence advice from other type(s) of tests.

4.2.2.2 Definitions

METHOD-SEQUENCE A $method\ sequence$ is a sequence of method calls, with each method call being a call to the functions from the system. In our framework, a method call is represented as a list of W_A(write action) and one I_A(invoke action). There is one W_A for each argument for the method, and one I_A for the receiver object if there is one. Thus, A *METHOD-SEQUENCE* is a sequence of method calls $(m_1, m_2, \ldots m_n)$, and for $i \in [1, n]$, we have $m_i = \{(W_A)^*(I_A)^+\}$

Sequence Advice Function

A sequence advice function, $advSeq$, applies advice on the sequences of function calls that may help setup parts of the system state that may not otherwise be accessible due to limitations in a particular testing technique.

We define $extractSeq\colon \mathbb{F} \to \mathbb{A}$ to extract *method sequence* from tests. Specifically, we extract sequences of *invoke* actions in tests stored in the framework to generate advice A_{Seq}. In addition to the *invoke* actions, the *write* actions on which they are dependent are also extracted to specify the function call and its arguments.

The method sequence advice, A_{Seq}, is used to control how the method invocations are ordered by a test generator. In Section 4.2.2.3 we explain the implementation details of an instantiation of $advSeq$ which inserts such sequences as test setup logic for tests produced by Symbolic PathFinder. In this instantiation we use the sequences as test prefixes, but they could also be inserted elsewhere in the test.

4.2.2.3 Combining System-level Random Concrete Execution Sequences with Unit-level Symbolic Execution Sequences

Random testing is usually fast, easy to scale up to large systems, capable of revealing software defects, and thus is considered as a cost effective testing technique [17, 21, 36, 53]. Gutjahr et al. [35] have also shown that random testing can be as effective as systematic testing techniques. Especially, with recent research advances in directed random testing, the tests generated by random approaches have fewer redundant and illegal test inputs [53]. However, there are cases when random approaches cannot reach some code guarded by

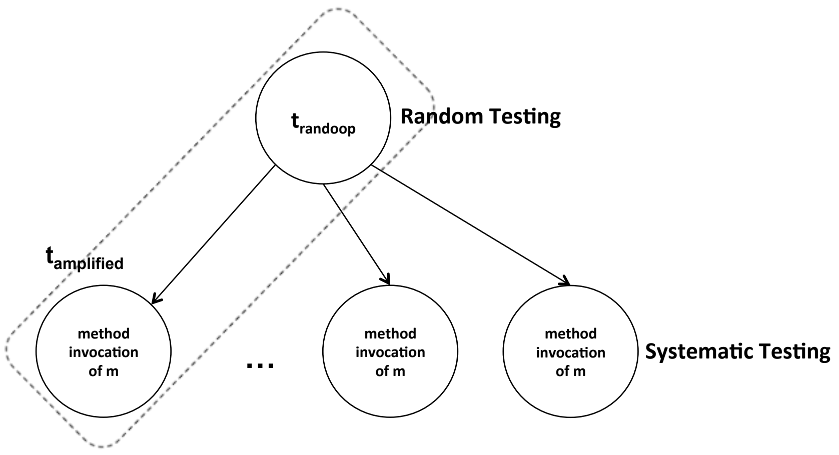


Figure 4.5: Relationship between $t_{randoop}$ in $T_{randoop}$ and T_{amp} in T_{amp}

specific predicates as we illustrated in Figure 4.3, in which the random approaches are blocked because of some particular input values needed.

Systematic testing such as SPF [56] and jCUTE [63], on the other hand, tends to be exhaustive and explores all possible behaviors of the SUT, but it is very expensive and suffers from the scalability issues. It also tends to focus on a particular part of the state space and results in less diverse test inputs. For example, Păsăreanu et al. [56] pointed out that SPF is general enough to be applied at various testing phases, but is more suitable for unit-level testing.

However, one problem with unit-level systematic testing based on a Symbolic Execution based approach is that it does not consider that valid inputs to a particular unit are constrained by the calling context of the unit. Consequently, having the systematic testing focus only on the unit-level would result in wasting time to generate data that may not appear in the context of the program execution. This is where our method sequence advice is particularly helpful. Directed random testing is fast and good at generating feasible method sequences that can be used to set up the state of the program. We conjecture that building on top of the feasible program state, systematic testing can take over and explore the full behavior of the unit under test.

We use SPF as an example of a systematic approach and Randoop as a random approach to explain the concept. As shown in Figure 4.5, $t_{randoop}$ refers to a test that Randoop generates at the system-level and we can base on it to set up the program state. Then, we can run systematic testing like SPF on the target method m and have it generate all possible argument lists to invoke m. Finally, Each pair of the ($t_{randoop}$, method invocation of m) is an amplified test t_{amp} in T_{amp}.

More specifically, this sequence advice function is:

$$extractSeq : RandoopTests \to Sequences,$$

$$gen : SPF \to SPFTests,$$

and

$$T_{amp} = advSeq((Sequences)(SPF))$$

Implementation We have discussed both test generation techniques in previous sections. However, one thing about Randoop that is worth mentioning again is that it is a directed random test generation technique. It executes the method sequences during its generation process, leaving out any sequences trapped by exceptions. Thus, with the passing tests that Randoop generates, we can trust the method sequences are valid to setup the environment for the target method under test.

We followed the process in Figure 4.6 to implement the advice. First, we use the test transformer described in Section 3.3 to transform the given Randoop test suite into the framework. Based on the common test representation of the Randoop tests, the $Sequence$ $Composer$ composes a method sequence in the Java code format for each test in the given

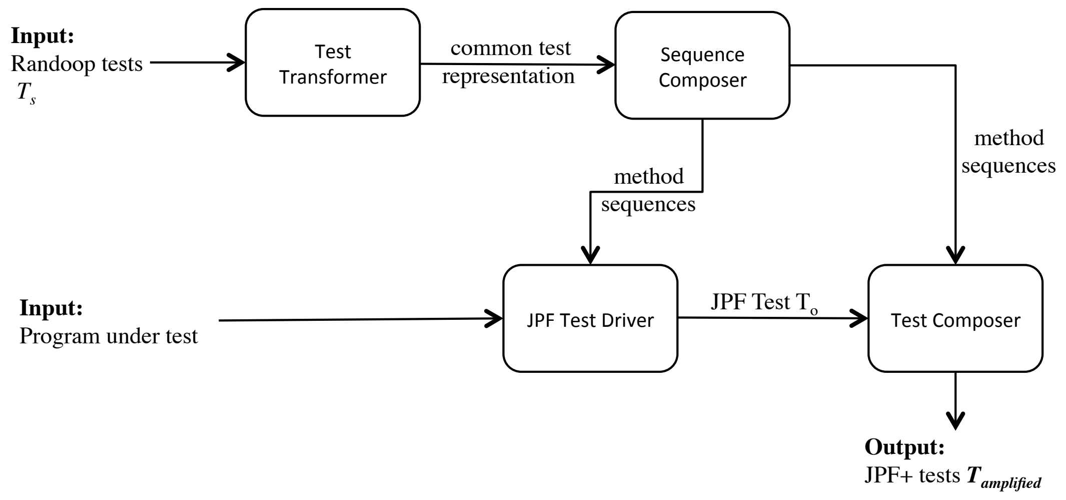


Figure 4.6: Process to Generate Amplified Test Suite using Method Sequence Advice

Randoop test suite. This step is particularly helpful when, for example, complex setup methods in the test class are present or if there are field values that a test refers to by name in tests. The transformer transforms a stand-alone method sequence after propagating the setup methods and field values for the current test.

Second, a SPF test driver uses the method sequence output by $Sequence\ Composer$ to set up state in the test driver before exploring the method. The test driver always has the sequence length set to 1 and generates SPF tests with one method invocation of m. Finally, the $Test\ Composer$ takes in the method sequence for the Randoop test suite and the method sequence for the SPF tests to output the set of tests for T_{amp}. As shown in the advice function, for each SPF test generated, there exists one test in T_{amp} and the test includes the setup sequence followed by the method invocation of m generated in the SPF tests.

For the evaluation in this section, we only worked on a limited number of tests and methods to explore the potential of the technique, handling them manually. However, we do see the possibility of automating the whole process.

4.2.3 Advice on Test Oracles

Oracle advice is an especially interesting advice type since automated oracle generation is still quite limited [8].

4.2.3.1 Motivation

The process of software testing relies on an oracle to determine if the system under test behaves correctly on a particular execution of a test case. Studies have shown that the effectiveness of a test suite decreases with oracle decay [62]. In spite of the importance of test oracles in successful software testing, it still remains a challenging area. Automated test generation techniques are getting very good at generating test inputs [46, 52, 73, 74, 76], but there is limited support when it comes to automating test oracles [8].

We attempt to address the test oracle problem by providing test oracle advice leveraging information from other testing tools or other type(s) of tests. For example, even though generating tests manually is costly and often inadequately done, the oracles in manual tests are valuable in that they capture the testers' understanding of the SUT and knowledge of what is important to be checked. This is valuable information that can benefit tests generated through automated tools.

4.2.3.2 Definition

ORACLE ADVICE FUNCTION

An oracle advice function, $advO$, applies advice on the tests that are performed on the SUT state resulting from its invocation.

We define $extractO\colon \mathbb{F} \rightarrow \mathbb{A}$ to extract oracle data from the tests stored in the framework. More specifically the oracle advice, A_O, is extracted from the *decision* actions, and the comparison function (f_{CMP}) they contain. For each oracle o extracted from the framework, a *scope*, s, is inferred to identify where the oracle can be used within the test. The scope can be inferred from the context $\mathbb{C}$.

For this instantiation of $advO$, the advice computed by $extractO$, $A_O = \{(o_1, s_1), (o_2, s_2), ..., (o_n, s_n)\}$, is used to enrich the set of test oracles that are inserted by a test generator. To ensure appropriate application of this advice, $advO$ only inserts oracles at method calls that match the scope associated with a piece of advice. In Section 4.2.3.3, we explain the implementation details of a concrete instance of the test oracle advice function.

4.2.3.3 Oracle Advice from Hand-Coded Tests to Others

e have discussed different aspects of hand-coded tests in the previous sections. In this section, we will be looking at another type of hand-coded tests that are written in adherence to the specifications of the programs.

A program specification is a technical contract for the program that describes what the system should achieve. There are different kinds of formal specifications, and we needed one that works with the Java language. So we consider in this section the formal specification language - JML (Java Modelling Language). JML is a behavioral interface specification language for Java [43]. It has many sample artifacts that come with the project and it has been used in plenty of research work. To be able to integrate with our framework, we manually translated the JML specifications to hand-coded tests in the JUnit framework. In Section 4.2.3.3, we describe how we derive the manual tests from the specification. The translated hand-coded tests will be the advice source. In the evaluation, we used Randoop tests as representative of all other types of tests, but note that all other types can benefit from this type of advice.

More specifically, the oracle advice function we implement in this section is:

$$extractO : Hand\text{-}Coded \rightarrow Oracles,$$

$$gen : \perp \rightarrow jUnitTests,$$

and

$$T_{amp} = advO((Oracles)())$$

There is an empty test generator $\perp$ in gen function, since the oracle advice function operates on an existing test suite.

Implementation First, there is only a small subset of JML specifications that we currently consider in the evaluation of our framework. Namely, we incorporate the following semantics: method specification clauses (*requires*, *ensures*), specification expression (\result), *model type declaration* and *Model Programs*. *ensures* is directly mapped to the assertions in the tests. *requires* is treated as a precondition of the checks that happen in the corresponding *ensures*, and the precondition is encoded as a *condition* in the related decision action. \result is interpreted as the return value of the method invocation. *model methods* in JML are methods that are declared to help in a specification, and they are treated as the helper methods in a test class. In other words, we declared helper methods in the test classes and invoke these methods in the assertions to help check the correctness of the implementation. The artifact we use in the evaluation in Section 5.5 uses the model program *JMLDouble* and *JMLDouble.approximatelyEqualTo* works as *assertEquals* with expected value, actual value, and a precision being its tolerance.

In JML, the program specification is specified as the pre- and post-conditions of methods, and class invariants. To take advantage of this rich in-line specification, we construct one manual test for each method, which includes a method call, and oracles checking on post-conditions after the the method invocation when the corresponding pre-conditions are satisfied.

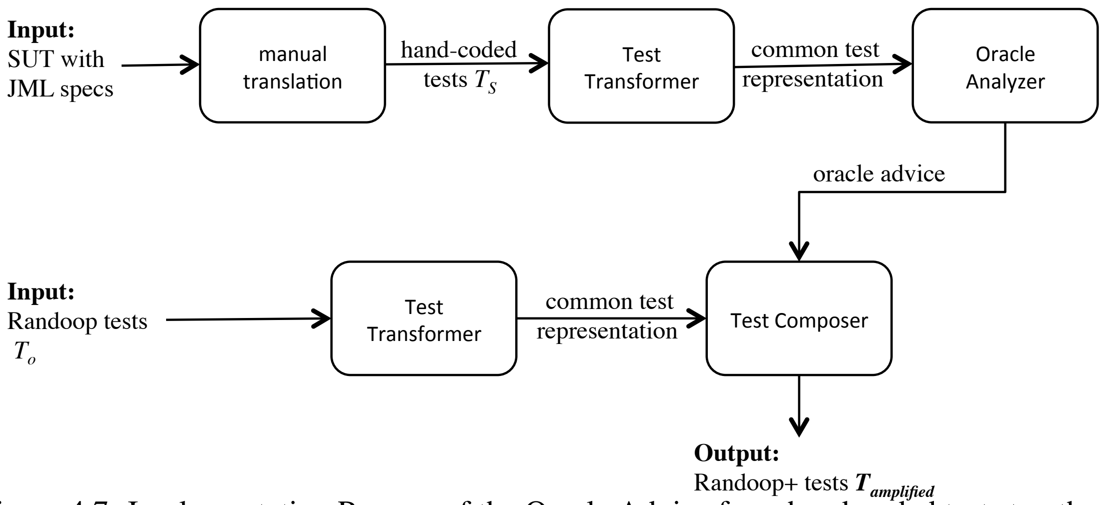


Figure 4.7: Implementation Process of the Oracle Advice from hand-coded tests to others

Figure 4.7 describes the process that we have implemented. The *oracle analyzer* takes in the hand-coded tests from the test representation framework, and generates the oracle advice. Since JML specifies pre- and post-conditions for methods, the manually translated hand-coded tests basically specifies a list of assertions following each method invocation. The *oracle analyzer* takes this advantage and generates the oracle advice for each method in the SUT. Figure 4.8 is an example showing the advice that the *oracle analyzer* generates. Note that in the oracle advice generated, we do not have any specific values, because the assertions in the hand-coded tests from JML specification are all generalized.

```
methodSignature: Complex.add(Complex arg1)
condition: arg1 != null
oracles:
        1. assertNotNull(add_return_value)
        2. assertEquals(arg1.realValue()+receiverObject.realValue(),
                     add_return_value.realValue(), 0.005)
```

Figure 4.8: Example To Show Oracle Advice Generated By Oracle Analyzer

The *test composer* will take the oracle advice generated by the *oracle analyzer*, the Randoop test suite from the framework, and output the amplified the Randoop+ tests T_{amp} with the oracles from the hand-coded tests. It performs oracle matching at the method level based on the method signature. The *oracle analyzer* generates oracle advice for each method in the SUT for which advice is available. Whenever the *test composer* comes across a method invocation in given Randoop test suite, it searches to see if a piece of advice is available for the method. If there is a match, it instantiates the oracle template with the concrete instance in the test. When the *test composer* goes through all method invocations in the test, it outputs the amplified test.

Chapter 5

Evaluation

In this Chapter, we perform an evaluation on the four concrete instantiations of the advice function introduced in the previous chapter.

5.1 Research Questions

The goal of this section is to provide a preliminary evaluation of advice functions on test inputs, sequences, and oracles, through an assessment of the effectiveness of the original test suites versus the amplified test suites with the advice. To evaluate the effectiveness, we measure code coverage in terms of line and branch coverage with Cobertura [16], and mutants killed score with Javalanche [61].

Research Questions

We developed two research questions that we aim to answer in the study. We evaluate the quality of the advice functions by checking whether value, in terms of coverage and mutant score, is added to the amplified test suites.

- RQ1: Does the advice functions add value in achieving higher coverage? We compare the line coverage and branch coverage achieved by the original test suite and the amplified test suite with advice function applied.

- RQ2: Does the advice functions add value in achieving better mutant detection capability? We compare the mutant score of the original test suite and the amplified test suite with advice function applied.

5.2 Artifacts

The study is performed primarily on the Java prototype of the ACCoRD framework developed at NASA for the formal specification and verification of state-based conflict detection and resolution algorithms [1]. The framework has 77 classes in total. We scoped our evaluation to the 16 classes (2468 LOC) that have hand-coded JUnit tests, which were needed for the study. The first three columns of Table 5.2 provide a brief characterization of these classes. For the evaluation in Section 5.3.2, we used Wheel Brake System (WBS with 212 LOC) and Altitude Switch (ASW with 295 LOC). In addition, for the study in Section 5.5.1, we use the Complex Number artifact (141 LOC).

5.3 Input Advice

5.3.1 From Hand-Coded Tests to Randomly Generated Tests

We mentioned before that Randoop tests may be limited by the tool's initial pool of input values. In this section we explore the effects of enriching that pool by applying input advice extracted from manual tests, which tend to encode input values deemed as valuable by the programmer. More specifically we explore the input advice function defined in

Section 4.2.1.3. To recall, $extractIn : ManualTests \rightarrow Inputs$, $gen : Randoop \rightarrow RandoopTests$, and $T_{amp} = advIn(Inputs)(Randoop)$. The scope for the $Inputs$ advice is computed at the class-level.

5.3.1.1 Experimental Design

To measure the quality of the generated Randoop tests in terms of the two research questions, we first generated T_{orig} (Randoop original configuration) and T_{amp} (Randoop+ test suite with original configuration + $inputAdvice$ function). Then we used Cobertura [16] to compute the line coverage and branch coverage and used Javalanche [61] to compute the mutant score of all test suites generated.

When calculating coverage, we enforced some special considerations. Randoop makes random choices while generating tests, based on the random seed used in the configuration. Sometimes a different random seed could make a difference in the test suite generated. To mitigate the effect from this randomness, we generated 10 test suites for each of the 16 ACCoRD classes. We picked at random 5 different seeds (0, 4, 7, 11, 13), and ran Randoop twice for each seed. Finally, we calculated the average coverage the 10 runs and reported the average. Also, time bound to generate tests are 10 seconds for 14 classes and 100 seconds for 2 classes. Given the size of the classes, we found that 10 seconds is a good time limit to achieve a consistent coverage. “IntentCriteria” and “Kinematics” exhibit non-terminating behavior which traps Randoop into an infinite loop during test generation. In Randoop’s configuration, the option “usethreads” can be configured to execute each test in a separate thread and kill tests when it is taking too long to finish. In our case, the “usethreads” flag needed to be turned on for the 2 classes which slowed down the test generation, and thus a longer time interval was given to make up the gap.

We made a couple arrangements when calculating mutant score. Despite the advances in getting more effective mutation tools, it is still very expensive to calculate mutant score for a target program. For example, for our experiment we would need to run 320 test suites of over 20 million tests. Also, the test suites have to be run against every mutated version of the SUT. With the understanding of the random side of the Randoop tool, we used the most common seed 0 that testers use to generate Randoop tests and performed our evaluation on these suites. The test suite generated with seed 0 is consistent with the average coverage that we achieved for validating RQ1, and this also assured us that these test suites are valid for us to use for the exploration of RQ2. Two out of the 16 classes are not included in this evaluation: "Priority" and "Units", because their Randoop tests depend on other tests in the suite, which causes Javalanche to calculate incorrect mutant scores. For the classes `Util` and `WGS84`, which had over $150,000$ tests, we only retained $10,000$ to control the cost of mutation evaluation.

5.3.1.2 Results

Research Question 1: Higher Coverage Achieved? Does the input value advice add value in achieving higher coverage?

Table 5.2 demonstrates the detailed results for a coverage comparison between T_{orig} (Randoop original configuration) and T_{amp} (Randoop+ test suite with original configuration + $inputAdvice$ function). As mentioned in Section 5.3.1.1, the results reflect the average of 10 Randoop runs for each class.

From Table 5.2, we can see that T_{amp} has better performance for 9 out of 16 classes. In particular, for the "SeparatedInput" class, T_{amp} achieved an increase of 10.2% for line coverage and 21.6% for branch coverage. Some specific strings, which are not in Randoop's initial pool of values, are needed to explore some methods in this SUT. Figure 5.1 is one such example. On average across all 16 classes, T_{amp} covers 36 more lines of code and 44 more predicates. The higher increase in predicate number than LOC can result when there is no matching else block for an if predicate. However, we can not underestimate

```
\n\n# Test\n\n  #Test = 3\n\nx = hello
\ny = 10\nz    = 10  [nmi]\ncol1 col2, Col3\n 1 2 3\n 4 5 6\n
```

Figure 5.1: Example of Specific String Needed in Randoop's Test Generation

the effect of the increase in the number of predicates covered. Covering more branches can sometimes bring more benefits if there is a potential bug or there is a big chunk of code not covered in that branch.

4 out of the 16 classes did not reflect any change in terms of coverage. These 4 classes all have very simple implementations without much interaction between method invocations, and they did not require any extra input values. There are 3 classes where T_{amp} ended up with a lower coverage: "GreatCircle", "IntentCriteria", and "SimpleProjection". The average difference was small. On average, T_{amp} covered 0.76 less branches for class "GreatCircle", 0.1 less branches for class "IntentCriteria" and equal numbers of line coverage. For class "SimpleProjection", T_{amp} covered 4.16 less lines of code and 0.384 less branches. When we extended the test generation time from 10s to 100s, "GreatCircle" and "IntentCriteria" achieved equal coverage for T_{orig} and T_{amp}.

However, "SimpleProjection" class illustrates the case that undesirable input values given to Randoop could potentially harm the test generation process instead. The more input values added to Randoop's pool, the more time Randoop would need to explore all combinations. When the advice provided is not desirable, it will just divert Randoop and take longer to explore the program states that it could have otherwise reached.

Research Question 2: Better Mutant Detection Capability? Does the input value advice add value in achieving higher mutant score?

Table 5.1 demonstrates the detailed results for the comparison of mutant detection capability between T_{orig} (Randoop original configuration) and T_{amp} (Randoop+ test suite with original configuration + $inputAdvice$ function). The findings are consistent with the coverage results presented above – overall, T_{amp} is more effective at killing mutants than Randoop (T_{amp} kills 178 mutants more than $RandoopTests$). The input advice helps to improve half of the classes, is neutral for four classes, and has a negative impact on three classes. In some cases, e.g., `SeparateInput`, the improvement is close to 25%, while the worst performance is for `SimpleProjection` with a decrease in mutants killed of 2%. In all cases, increasing the test generation time would improve the performance of Randoop only when the advice was applied. Thus, Randoop tests with the advice from manual tests are more capable of killing more mutants than those without, which in turn indicates better bug detection capability.

Class Name	**# of Mutants**	**# of Tests**		**Mutant Score (%)**	
		T_{orig}	T_{amp}	T_{orig}	T_{amp}
AircraftState	803	10300	9940	47.45	55.17
CDII	172	10840	10353	66.28	66.28
CDSI	175	7269	9284	71.43	71.43
GreatCircle	446	15141	17246	79.82	79.37
IntervalSet	226	9267	11114	51.77	53.54
IntentCriteria	24	210	320	83.33	87.50
Kinematics	526	4146	4282	57.41	58.94
Plan	116	12024	11693	69.57	67.83
PlanCore	912	8589	7843	48.25	50.66
SeparatedInput	292	9788	8806	50.34	74.32
SimpleProjection	262	8767	8012	76.72	74.43
Util	324	10000	10000	64.81	71.30
Velocity	102	13442	7434	98.04	98.04
WGS84	114	10000	10000	70.18	70.18
Total	**4493**	**129783**	**126236**	-	-
Average %	-	-	-	**59.50**	**63.45**

Table 5.1: Mutant Kill Score Comparison between Randoop (T_{orig}) and Randoop with Input Advice (T_{amp}).

Class Name	**LOC**	**# of predicates**	**# of Tests**		**Line Coverage %**		**Branch Coverage %**	
			T_{orig}	T_{amp}	T_{orig}	T_{amp}	T_{orig}	T_{amp}
AircraftState	356	152	9517.8	10641.5	72.5	73	69	69.9
CDII	71	36	10530.3	11793.2	85	85	77	77
CDSI	85	34	7581.6	8394.6	91	91	79	79
GreatCircle	118	38	17547.5	18191.2	98	98	93.8	93.6
IntervalSet	150	86	8541.6	10332.3	87.3	88.4	76.1	80
IntentCriteria	18	2	413.8	299.1	100	100	90	85
Kinematics	270	102	4345.6	4428.9	79.6	80.1	75.2	75.5
Plan	81	26	12909.1	12434.3	93	93	69.8	73
PlanCore	391	268	7744.2	7209.4	75.8	77	64	66.2
Priority	58	2	76225	79380.7	94	94	50	50
SeparatedInput	174	110	19074.9	18495.4	75.2	85.4	63.2	84.8
SimpleProjection	104	48	8144.3	7828.8	94.2	93.8	87.4	86.6
Units	428	88	30753.7	34331.8	81	82.6	82.2	88.9
Velocity	32	2	10677.7	8362.8	94.2	94.8	70	80
Util	100	94	157605.9	157210.4	87	89.2	84.6	87.6
WGS84	27	-	249961.7	244468.8	92	92	-	-
Total	**2468**	**1088**	**631574.7**	**633803.2**	-	-	-	-
Average %	-	-	-	-	**81.97**	**83.44**	**73.02**	**77.05**

Table 5.2: Coverage of Randoop (T_{orig}) and Randoop with Input Advice (T_{amp}).

5.3.2 From Systematically Generated Tests to Randomly Generated Tests

Similar to Section 5.3.1, in this section we explore the effects of enriching that pool by applying input advice extracted from SPF tests, which tend to encode input values deemed as valuable by the programmer. More specifically we explore the input advice function defined in Section 4.2.1.4. To recall, $extractIn : SPFTests \rightarrow Inputs$, $gen : Randoop \rightarrow RandoopTests$, and $T_{amp} = advIn(Inputs)(Randoop)$. The scope for the $Inputs$ advice is computed at the class-level.

5.3.2.1 Experiment Design

For this study, we used Wheel Brake System (WBS) and Altitude Switch (ASW). SPF does not work well with the 16 classes in ACCoRD, because of the complex objects needed as method arguments. Given a 5-hour test generation time limit, SPF is capable of generating tests with average method sequence of length 0.9 for the 16 classes. WBS is used to provide safe retardation of an aircraft during taxi and landing and in the event of an aborted take-off. It has one class and 212 lines of source code. ASW is a synchronous reactive component from the avionics domain with one class and 295 lines of source code. Both artifacts are within the applicability of both SPF and Randoop, which makes them qualified artifacts in the experiment setting in the evaluation of the input advice from SPF tests to Randoop tests.

The procedure is quite similar to the one in Section 5.3.1. To measure the quality of the generated Randoop tests in terms of the two research questions, we first generated T_{orig} (Randoop original configuration) and T_{amp} (Randoop+ test suite with original configuration + $inputAdvice$ function). Second, we used Cobertura [16] to compute the statement coverage and branch coverage for all generated test suites. Finally, we used Javalanche [61] to compute the mutant score of all test suites.

To calculate the coverage for both WBS and ASW, we also ran Randoop with 5 different seeds(0, 4, 7, 11, 13). However, both WBS and ASW are much less complicated compared to the 16 classes in ACCoRD. Given 5 seconds of time, Randoop was able to reach consistent coverage results for all seeds chosen. All the test suites used in this section for both coverage and mutant score calculation are generated with random seed of 0.

5.3.2.2 Results

In this section, we examine the results for the two research questions.

Research Question 1: Higher Coverage Achieved? Does the input value advice add value in achieving higher coverage?

From Table 5.3, we can identify a coverage increase in WBS, but ASW had the same coverage before and after the advice was deployed. We now explore the reasons for these different outcomes. In WBS, we identified constraints in the program for which Randoop failed to provide the needed input values. Figure 5.2 shows one of such examples that we identified in WBS. For the predicate at line 1 to be true, var1 must be 3. However, Randoop does not to have 3 in its initial pool of values. JPF-symbc, on the other hand, was able to solve the constraints in these classes and eventually generated tests with the desired input values. With these inputs provided as advice, Randoop was able to overcome the hurdle and went on with the test generation reaching such statements. For ASW however, Randoop was able to cover all the reachable code with its initial pool of values and thus the input advice from JPF-symbc tests were not able to help Randoop in this case.

Table 5.3: Coverage Comparison between Randoop and Randoop with JPF Input Advice

Subject	**# of Tests**		**Line Cov**		**Branch Cov**	
	T_{orig}	T_{amp}	T_{orig}	T_{amp}	T_{orig}	T_{amp}
WBS	6012	9618	67	82	50	68
ASW	4007	3867	88	88	70	70

```
if (((var1 >= 3) && (var1 < 4))) {
    WBS_Node_WBS_BSCU_Switch2 = 3;
} else {
    WBS_Node_WBS_BSCU_Switch2 = 4;
}
```

Figure 5.2: Example to demonstrate the importance of input values

Research Question 2: Better Mutant Detection Capability? Does the input advice add value in achieving higher mutant score?

Table 5.4 provides the information about the mutant detection capabilities of these test suites. From Table 5.4, we can see that the mutant detection capabilities are consistent with the coverage of the test suites. WBS test suites did manage to obtain a higher mutant score with the advice. This is reasonable considering the extra percentage of the code that was covered after the $inputAdvice$ function applied. ASW test suite remained the same in terms of mutant score, same as with the coverage.

Table 5.4: Mutant Detection Capability between Randoop and Randoop with JPF Input Advice

Subject	**# of mutants**	**# of Tests**		**Mutant Score**	
		T_{orig}	T_{amp}	T_{orig}	T_{amp}
WBS	264	6012	9618	56.44	80.30
ASW	265	4007	3867	89.81	89.81

5.3.2.3 Synthetic Program Snippet

As discussed in Section 5.3.2.2, method input values extracted from JPF-symbc tests were able to provide valuable advice to Randoop in cases where Randoop got stuck because of the constraints in a predicate. However, WBS did not reveal all the potential that this advice is capable of providing.

We illustrate the greater potential of the input advice function in the code shown in Figure 5.3. The class *testclass* has one public method waiting to be tested: *testme*. The method takes 2 integer arguments, which are needed for comparison in 2 equality constraints inside the method. Only when *x==1048576* and *equalityPredicate==7* will the bug be detected at line 11.

To test the method *testme*, if we use a random testing approach, whether the bug can be detected or not entirely depends on the method input values that the random approach has at hand. Take Randoop for example, it will not be able to cover any code inside the *if* predicate at line 4, and thus will leave the bug undetected. On the other hand, if we use a systematic testing approach like Symbolic Execution, the loop construct at line 6 will stop the approach from exploring all the program states because of the infinitely long execution paths that could possibly be generated. If we choose to perform a bound exhaustive approach, we will need a depth of over 1048576 to reveal the buggy code. Take JPF-symbc for example, it does not finish executing within 1-hour limit for a bound of 1048576. Given that this is a small example with one method under test, we set a small window of 1 hour to let it finish. If the bound is smaller, it will finish sooner without revealing the bug, but generating the desired input values.

In this case, a better alternative would be to advise Randoop with JPF method input values. JPF-symbc generates method input values of both 1048576 and 7 when the bound is set to 5 and finishes after 3 seconds. And on the other hand, it is easy for Randoop to

```java
public class testclass {
    int index = 0;
    public void testme(int x, int equalityPredicate){
        if(x==1048576){
            int i = 0;
            while(i<1048577){
                i++;
                increaseIndex();
                if(helper(equalityPredicate)){
                    //divide by 0 when index is 1048576
                    int temp = index/(index-x);
                }
            }
        }else{
            //everything is fine here
        }
    }

    private void increaseIndex(){
        index += 1;
    }

    private boolean helper(int equalityPredicate){
        if(equalityPredicate == 7)
            return true;
        else
            return false;
    }
}
```

Figure 5.3: Example to demonstrate the importance of input values

execute with concrete values inside the method. With the help of the input advice, T_{amp} detects the bug in 1 second.

5.3.3 Discussion on Input Advice

In Section 5.3.1 and Section 5.3.2, we have applied the advice at the class level. In other words, a literal extracted from a given class is used as input advice only to methods of that class for test generation. There are indeed other granularities at which this advice can be applied, for example the package level or the method level.

Theoretically, advising Randoop with the same set of input values at different levels should eventually converge to the same results, but it may take different amount of time as it enumerates all *(scope, values) pairs* of the test method input values. We did experience better results running at the class level than running at the package level. This is reasonable, because at a package level Randoop needs more time to enumerate all possible values for the scope of $package$, when some method input values are actually intended for methods of one particular class.

We anticipate that at the method level, we will achieve same coverage and mutant score as that of the class level within a shorter period of time and less tests. However, Randoop tool does not currently support this feature, so we did not validate this conjecture.

Robinson, et al. conducted similar experiments in [59], where they experienced better results at the package level, because there were cases where literals from one class were needed to invoke methods in another class. However, they extracted input values from the SUT to guide Randoop. Such input values do not necessarily reflect their needs for method arguments. On the contrary, the input values we extracted from the hand-coded tests or JPF tests are more targeted and are designed by the testers to be used as the method arguments. We conjecture that when these values are more targeted, it is better to restrict them to the level that they are from.

5.4 Sequence Advice

5.4.1 Combining System-level Random Concrete Execution Sequences with Unit-level Symbolic Execution Sequences

SPF tests are generally effective at generating inputs for primitive types to cover most paths in a target method. When a target method execution depends on an object that requires several method invocations to be built, however, SPF tends to struggle due to the exhaustive nature of its exploration of the program space. Randoop on the other hand tends to quickly build tests with long sequences of method invocations, which can often lead to the creation of interesting object structures. In this section we explore whether we can leverage the strengths of Randoop tests to set the state of the program such that SPF can generate tests with improved coverage and mutant scores on a set of methods identified as of interest by the developer. More specifically we explore the sequence advice function defined in Section 4.2.2. To recall, $extractSeq : RandooTests \rightarrow Sequences$, $gen : SPF \rightarrow SPFTests$, and $T_{amp} = advSeq(Sequences)(SPF)$.

5.4.1.1 Experimental Design

To perform this study we consider a scenario where a tester with access to SPF and Randoop is trying to improve the coverage of some methods. Given the large number of potential methods to target, we decided to restrict our attention to the two largest classes in ACCoRD with non-static methods (`AircraftState` and `PlanCore`).

First, we ran Randoop at the system-level. For the class that the target method is in, we ran Randoop for 10 seconds. This served as the advice source test suite from which we extract sequence advice. Then we also try to run SPF at the system-level. When running SPF, we get the longest possible method sequences given a time limit of 5-hour. SPF generation for "PlanCore" class went beyond the 5-hour time limit when the sequence length was set to 2, and generation for "AircraftState" fell into an error of "java.lang.RuntimeException: ## Error: SYMBOLIC IREM not supported" when the sequence length was set to 2. Thus, we generated SPF test suites for class "PlanCore" and "AircraftState" of length 1. Table 5.5 shows the profile of both the Randoop test suite and the JPF test suite generated for the two classes used in the evaluation. Randoop is able to generate long sequences within a very short period of time. For the advice to be applicable and valuable, SPF and Randoop tests must be able to reach the target methods but not fully cover them. Within the two target classes, we selected the three largest methods that met the previous criteria. Sequence advice was then extracted from the generated Randoop tests, and provided to SPF to guide its test case generation process by acting as prefixes to any exploration.

Table 5.5: Profile of Randoop and JPF Test Suites Generated

Class Name	**Time(s)**		**# of tests**		**Avg Seq Len**	
	JPF	**Ran**	**JPF**	**Ran**	**JPF**	**Ran**
AircraftState	30	10	53	8966	1	15
PlanCore	874	10	548	8227	1	21

5.4.1.2 Results

In this section, we examine the results obtained for the research question we raised above.

Research Question: Higher Coverage Achieved? Does the method sequence advice add value in achieving higher coverage on a the target methods using the sequence of method invocations from the Randoop tests to set up the desired program state?

Table 5.6 shows the results for the coverage metrics of the six methods before and after the advice is deployed. On average, we identified an average increase of 49% in line coverage and 41% in branch coverage. We can see that the method sequence advice function was able to increase the coverage of the target methods to varying extents (from 0% to 72.43%), and for some methods the advice function was not able to achieve full coverage on the target method. This is because some of the predicates inside the code are dependent on referenced values of the pre-defined concrete state. And in SPF, once the one variable is set to be concrete, it will stay as concrete during its execution. Thus, part of the code in these methods that is directly or indirectly dependent on referenced values was not covered.

Class.Method	# of lines	# of branches	# of tests		Line Cov %		Branch Cov %	
			T_{orig}	T_{amp}	T_{orig}	T_{amp}	T_{orig}	T_{amp}
AircraftState.find	7	5	1	1+14	28.57	100	60	100
AircraftState.add	22	9	1	1+4	31.82	90.91	22.22	77.78
AircraftState.predLinear	8	3	4	4+0	87.5	87.5	66.67	66.67
PlanCore.gsSmooth	22	11	5	5+7	36.36	86.36	45.45	81.82
PlanCore.timeshiftPlan	14	10	2	2+70	28.57	100	30	100
PlanCore.getIndex	14	13	1	1+11	64.29	92.86	61.54	84.62
Totals	**87**	**51**	**14**	**14+106**	-	-	-	-
Average %	-	-	-	-	**42.53**	**91.95**	**45.10**	**86.27**

Table 5.6: Coverage Comparison between SPF (T_{orig}) and SPF with Sequence Advice (T_{amp}).

Research Question: Higher Mutant Detection Capability Achieved? Does the method sequence advice add value in achieving higher mutant kill score on the target methods with the application of the sequence advice function?

Since SPF tests' oracles are weak, the small increases in mutation scores were expected (the amplified test suite killed 38 additional mutants). Table 5.7 shows the results for the mutant score metrics of the six methods before and after the advice is deployed. On average, we identified an average increase of 20% with the application of the advice function. It was interesting that most of the additional mutants killed came from *PlanCore.timeshiftPlan*. A closer analysis revealed that the original test suite could not reach the code within the nested loops in this method, where several mutants resided. Once that code become reachable due to the sequence advice, the SPF tests were able to reveal 21 of the 28 seeded mutants because of exceptions like *NullPointerException*, *ArrayIndexOutOfBoundException* or having program execution trapped in an infinite loop.

For method *"PlanCore.timeshiftPlan(int, double)"* shown in Figure 5.4, there are 9 tests which reached the "if" predicate at line 7. However, in all 9 cases, the argument

Class.Method	**# of**	**# of tests**		**Mutant Score %**	
	mutants T_{orig}	T_{amp}	T_{orig}	T_{amp}	
AircraftState.find	13	1	1+14	7.69	7.69
AircraftState.add	36	1	1+4	0	0
AircraftState.predLinear	17	4	4+0	11.76	11.76
PlanCore.gsSmooth	47	5	5+7	0	14.89
PlanCore.timeshiftPlan	28	2	2+70	0	75
PlanCore.getIndex	47	1	1+11	8.51	27.79
Totals	**188**	**14**	**14+106**	-	-
Average %	-	-	-	**3.72**	**23.94**

Table 5.7: Mutant Score Comparison between SPF (T_{orig}) and SPF with Sequence Advice (T_{amp}).

"double st" was not set right for it to reach the code inside the "if" predicate. On the other hand, the SPF tests can not set up a program state with "numPts" attribute greater than 0. And the "if" predicate at line 2 ensures that the local variable $start \geq 0$. Thus, none of the JPF tests can reach the code wrapped inside the "for" loop at line 3 and the "if" predicate at line 6. However, when we extract the method sequence from Randoop tests to help SPF to get inside the "for" loop at line 3 and the "if" predicate at line 6, SPF was able to generate tests that cover all the code in the method.

Also, for this method, not only did the advice manage to increase the coverage of the target method, it also revealed a *NullPointerException* existing in the code. The exception happens at line 14 with *"java.lang. NullPointerException: Calling 'time()D' on null object..."*. The code first calculates the larger value of two variables *start* and *lastOrig*, and then uses the larger value as an index to retrieve data from *points*. However, for the given test case, there is only 1 element in *points* at index 0. When *Math.max* returned 1 as the larger value, *points[1]* is null and it resulted in *NullPointerException* when trying to call *time()*. Not understanding the logic of the code, it is hard to explain what caused the *NullPointerException* or how to fix it though.

```
public void timeshiftPlan(int start, double st) {
    if (start < 0) start = 0;
    for (int i = start; i < numPts; i++) {
        setTime(i, points[i].time() + st);
    }
    if (start < numPts) {
        if (st < 0.0) {
            int lastOrig = start-1;
            if (lastOrig < 0)
                lastOrig = 0;
            double beforeTime = points[lastOrig].time();
            double cTime = points[Math.max(start,lastOrig+1)].time();
            while (cTime <= beforeTime && start < numPts) {
                removeWithRecord(start);
                cTime = points[start].time();
            }
        }
    }
}
```

Figure 5.4: Example target method with NullPointerException

5.4.1.3 Discussion on Combining Random and Systematic Approaches

There are a couple of points that we want to discuss further on the proposed approach which is a hybrid of Symbolic Execution and Random Concrete Execution.

First, the underlying reason that this combination can work is that the Random Concrete Execution at the system level is able to provide us with rich heap that can set up the interesting program state for symbolic execution to keep working on the target method. For the particular representative approach that we use in the evaluation - Randoop, it is a good candidate for the random execution at the system level because Randoop tends to build long method sequences with repetitive method calls. The long method sequences make a higher probability that a rich heap can be constructed. However, if the assumption fails to hold, Randoop may not be an ideal candidate to use.

Second, when generalized, the approach can be defined as executing the system concretely with Randoop for a given period of time until it struggles with a particular unit and then, SPF takes in the concrete method sequences generated by Randoop and exhaustively explores the unit under test based on the concrete program state set up by Randoop. The combination of running SPF first and then Randoop for method sequence generation, however, was not effective. We investigated running JPF-symbc first generating the longest possible method sequences given a time and memory limit, and then we had Randoop incorporate all those method sequences. We found, however, that SPF gets stuck with short sequences, and the sequences generated did not provide much value in terms of setting up a complex program state needed for Randoop.

Finally, another potential direction to investigate is to restrict the program state based on the pre-conditions of the MUT to improve the precision of symbolic execution on the unit-level as proposed by Păsăreanu, et al. in [56]. This would require for our framework to encode those and develop advice that use that information to instantiate the concrete program state that can be used for the unit-level symbolic execution.

5.5 Oracle Advice

5.5.1 Oracle Advice from Hand-Coded Tests to Others

Randoop tests can include a variety of regression oracles on method return values and checks against a small set of predefined contracts like *Symmetry of equality* and *Equals to null*. In this section we explore whether oracles extracted from manual tests that may encode a developer's understanding of the particular code specifications can enhance the fault detection of Randoop tests. More specifically we explore the oracle advice function defined in Section 4.2.3.3. To recall, $extractO : Hand\text{-}Coded \rightarrow Oracles$, $gen : \perp \rightarrow jUnitTests$, and $T_{amp} = advO(Oracles)()$.

5.5.1.1 Experimental Procedure

Our first attempt to explore this advice targeted the ACCoRD framework where we quickly faced a significant challenge. We found that advice extracted from the manual tests tended to be too limited in quantity and too specific to be widely applicable across the Randoop tests. For example, most manual tests made a sequence of invocations with a series of inputs and then checked for equality between a particular variable value against some expected value. Such specific advice could only be safely applied to Randoop tests whose prefix match the manual test sequence of invocations and input values; any relaxing of this matching may render the oracle inappropriate. One important lesson we learned from this experience, is that the more general oracle advice may be more broadly applicable across tests of different types. Within that context, we decided to explore tests that use more abstract oracles. More specifically, we targeted a sample class that is used in the learning materials of Java Modeling Language [13] and that comes with simple annotations such as methods pre- and post-conditions. Given the small subset of JML semantics we support currently, we chose $Complex\ numbers$ which supports arithmetic operations on complex numbers like $Rectangular$ or $Polar$. There are 4 classes in it with 141 LOC.

First, we transformed the program specification in JML into hand-coded tests. JML encodes pre- and post-conditions for each method. We hand-coded one test for each method in the classes, serving as the oracle template for the method. And this is our source test suite where we extract advice from.

Second, we generated Randoop tests for the 4 classes. At the very beginning, we generated Randoop tests just with random seed 0, but we discovered that the same type of mutants are sometimes killed and other times alive. Given the randomness of the Randoop tool, we generated 5 different test suites using 5 different seeds and use them all in the experiment. This generates the original test suite T_{orig}.

Third, we passed both hand-coded tests and Randoop tests to our $oracleAdvice$ function to generate the T_{amp}. Finally, using Javalanche, we computed the mutant score for both T_{orig} and T_{amp} and report the results for a comparison of the two for the T_{orig} generated for each of the 5 random seeds.

5.5.1.2 Results

Since we only aim to amplify the oracles in the test suite, we do not modify the test inputs. Coverage between the original tests and amplified tests remain the same. Thus, we only explore the improvement in mutant kill score in this section.

Research Question 2: Better Mutant Detection Capability? Does the oracle advice add value in achieving higher mutant score?

Table 5.8 shows the results we have for the comparison of the mutant detection capabilities. First, we can see that the mutant score does increase for T_{amp} with the oracle advice applied from the hand-coded tests on top of the Randoop test suite. This is what we expected since we are adding the oracles and the mutant score should increase or at least stay the same. There are 2 reasons for the mutant score for T_{amp} not to be 100%. First, after manual inspection, we found that there are 8 mutants that are equivalent. For example, 4 mutants operate on a local variable, but there are assignments to that local variable on all paths in the rest of the method. Thus, the mutants don't really change the semantics of the program, which we consider as equivalent. Second, the test inputs are not comprehensive to cover the mutant or it covers the mutant but does not provide specific values to kill the mutant.

Another interesting point about the result is the randomness of the test suite. The mutant score for $T_{randoop}$ varies depending on which seed is used for test generation, even though the number of covered mutants are similar. On the other hand, the mutant score for

Artifact	Seed	# of tests	# of mutants	Covered mutants	Mutant Score %	
					T_{orig}	T_{amp}
	0	1731		122	78.62	83.21
Complex	4	1256		124	67.94	85.50
	7	1453	131	122	76.34	83.21
numbers	11	1134		124	68.70	85.50
	13	1112		120	61.83	81.68
Average %	-	-	-	**122.4**	**70.69**	**83.82**

Table 5.8: Mutant Detection Capability of RandoopTests and T_{amp} with the Oracle from Hand-coded Tests

T_{amp} stays stable considering the number of mutants covered. We will explain the reasons for this using the example shown in Figure 5.5.

```
1 public Complex sub(Complex b) {
2     return new Rectangular(this.realPart() - b.realPart(),
3           this.imaginaryPart() - b.imaginaryPart());
4 }
```

Figure 5.5: Example to show the randomness in mutant score that exists in $T_{randoop}$

One of the mutants Javalanche generates for this method is at line 2 – “Replace arithmetic operator (Replace DSUB operator with DADD operator)”. In this case when the return value is an object of type “Complex”, the types of errors Randoop checks for include *Reflexity of equality*, *Symmetry of equality*, *Equals to null*, *No null pointer exceptions* and *Equals-hashcode*. All these checks are not capable of killing the mutant. Only when Randoop by chance calls “double realPart()” on the returned “Complex” object and checks the return value will the mutant be killed. However, tests in T_{amp} ensure that the $realPart()$ is checked every time the method call is made, which is why the results are more consistent.

5.6 Threats to Validity

Our findings are subject to several threats to validity, mostly due to the preliminary stage of this work. First, we limited our study to a small number of test suite types and test case generation tools, all of which are represented in a format that we could quickly manipulate for advice extraction or generation. We argue but are yet to show in practice the degree of generality of the test representation language, and the cost-effectiveness for advice extraction and generation on more diverse suites such as those aiming to check a GUI or stress a system's resources. Second, the artifact that we have used serves as a solid starting point, but larger programs with larger suites will let us analyze, for example, whether the approach scales and whether the limitations identified at the end of Chapter 3 manifest in practice. Third, there are several other metrics that we did not consider and that should be part of a larger evaluation such as the cost of advice extraction and generation. Fourth, we study very particular instantiations of the advices and in constrained contexts, which limits the generality of the findings but helped us to observe the advice's effectiveness at this early stage of the project.

Chapter 6

Conclusions and Future Work

In this work, we defined a Test Case Language (TCL) which can be used to derive a common test representation for tests of varying formats. The test case language specifies the components of a test in terms of invoke action, write action and decision action. We demonstrated the applicability of the language through a series of examples covering tests of different formats. We also presented a framework for transforming tests of different representations into TCL. Two types of tests are incorporated into the framework currently - tests specified in Java source code and DUTs in XML format. However, other types of tests can be easily incorporated.

Finally, once tests are in a common representation and can be analyzed, we showed how valuable advice can be leveraged from one type of tests to amplify other type(s) of tests. We defined five advice functions - two on test input values, one on method call sequences, and two on test oracles. We incorporated three types of tests generated through different techniques including hand-coded tests, randomly generated tests from Randoop and systematically generated tests from JPF-symbc and operated the five advice functions on these three types of tests. We evaluated each advice function by checking the value of the amplified test suite with respect to code coverage and mutant kill score. Results show that the amplified test suite derived from the advice functions has improved value compared to the original test suite before the advice functions applied.

From our experience working with the framework and analysis discussed in this thesis, we envision several directions of future work.

First, we want to extend the current framework to incorporate more varieties of tests, non-functional tests for example. This will help us to improve the applicability of the framework, and incorporate more testing data for further analysis. Second, we want to broaden the the family of advice to add advice in other directions. For example, we could try to remove redundant tests based on the abstract test cases generated by JPF. JPF outputs abstract tests in path constraints and each path constraint reflects one partition out of the total input space. By leveraging this information, we can partition the concrete tests and remove the redundant tests which fall into the same partition. Third, we plan to study the limitations of the advice and the possibilities of providing bad advice. We explored four concrete advice functions in this thesis and assessment shows that the four advice functions do add value in the amplified test suites. However, we've also had experience with a bad advice function when the amplified test suite did not manifest any added value compared to the original test suite. As part of our future work, we plan to study this thread in more detail. Fourth, we want to explore other dimensions to take advantage of the testing data available in the framework. We want to bring about new metrics based exclusively on the test representation to evaluate the tests besides the commonly used code coverage and mutant kill score. Test prioritization is also a possible direction, to prioritize the tests generated through different techniques by maximizing the diversity of the tests - for example, not only to maximize the methods invoked, but also to maximize the diversity in the memory locations being checked. We believe that a corpus of diverse tests has the potential to be a rich source of information for the whole validation cycle.

Bibliography

[1] ACCoRD. `http://shemesh.larc.nasa.gov/people/cam/ACCoRD/`.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[3] Shay Artzi, Michael D. Ernst, Adam Kie. Zun, Carlos Pacheco Jeff, and H. Perkinsmit Csail. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *In M-TOOS*, page 2006, 2006.

[4] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Softw. Eng. Notes*, 14(8):210–218, November 1989.

[5] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1243–1251. VLDB Endowment, 2007.

[6] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd inter-*

national conference on Very large data bases, VLDB '07, pages 1243–1251. VLDB Endowment, 2007.

[7] Kent Beck. jUnit. `https://github.com/KentBeck/junit`.

[8] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 219–227, New York, NY, USA, 2000. ACM.

[11] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1535–1544, New York, NY, USA, 2010. ACM.

[12] T. Y. Chen, Pak-Lok Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. *IEEE Trans. Softw. Eng.*, 29(7):577–593, July 2003.

[13] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proceedings of the 16th European Conference*

on Object-Oriented Programming, ECOOP '02, pages 231–255, London, UK, UK, 2002. Springer-Verlag.

[14] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

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

[16] Cobertura. `http://cobertura.sourceforge.net/`.

[17] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.

[18] T. Daboczi, I. Kollar, G. Simon, and T. Megyeri. Automatic testing of graphical user interfaces. In *Instrumentation and Measurement Technology Conference, 2003. IMTC '03. Proceedings of the 20th IEEE*, volume 1, pages 441 – 445, may 2003.

[19] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[20] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic load testing of web applications. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 57–70, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, july 1984.

[22] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance.* Bonston: Addison-Wesley, 1999.

[23] S. Elbaum, Hui Nee Chin, M.B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *Software Engineering, IEEE Transactions on*, 35(1):29 –45, jan.-feb. 2009.

[24] EMMA. `http://emma.sourceforge.net`.

[25] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, January 1996.

[26] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.

[27] Chen Fu, Mark Grechanik, and Qing Xie. Inferring types of references to gui objects in test scripts. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[28] Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 25–36, New York, NY, USA, 2010. ACM.

[29] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming lan-*

guage design and implementation, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[30] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *SIGPLAN Not.*, 10(6):493–510, April 1975.

[31] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Trans. Softw. Eng.*, 9(6):686–709, November 1983.

[32] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Wolfgang Grieskamp, Nikolai Tillmann, Colin Campbell, Wolfram Schulte, and Margus Veanes. Action machines - towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 72–82, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA, 2012. ACM.

[35] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 25(5):661–674, September 1999.

[36] Dick Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 1–9, New York, NY, USA, 2006. ACM.

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

[38] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009.

[39] Jumble. `http://jumble.sourceforge.net/`.

[40] K. Karhu, T. Repo, O. Taipale, and K. Smolander. Empirical observations on software testing automation. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 201 –209, april 2009.

[41] Bogdan Korel. Automated test data generation for programs with procedures. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '96, pages 209–215, New York, NY, USA, 1996. ACM.

[42] M. Landhausser and W.F. Tichy. Automated test-case generation by cloning. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 83 –88, june 2012.

[43] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[44] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In

Proceedings of the 11th Asia-Pacific Software Engineering Conference, APSEC '04, pages 284–291, Washington, DC, USA, 2004. IEEE Computer Society.

[45] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005.

[46] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 771–774, Washington, DC, USA, 2007. IEEE Computer Society.

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

[48] Tim Miller and Paul Strooper. A framework and tool support for the systematic testing of model-based specifications. *ACM Trans. Softw. Eng. Methodol.*, 12(4):409–439, October 2003.

[49] Ivan Moore. Jester- a JUnit test tester. In Proc. of 2nd XP, pages 84–87, 2001.

[50] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[51] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[52] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *In 19th European Conference Object-Oriented Programming*, pages 504–527, 2005.

[53] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference*

on Software Engineering, ICSE ’07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE ’12, pages 33:1–33:11, New York, NY, USA, 2012. ACM.

[55] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE ’10, pages 179–180, New York, NY, USA, 2010. ACM.

[56] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA ’08, pages 15–26, New York, NY, USA, 2008. ACM.

[57] Quilt. `http://quilt.sourceforge.net/`.

[58] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, April 1985.

[59] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE ’11, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.

[60] G. Rothermel, S. Elbaum, and H Do. *Software Infrastructure Repository*. http://cse.unl.edu/ galileo/php/sir/index.php, Jan, 2006.

[61] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM.

[62] David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *ICST '11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, March 2011.

[63] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

[64] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. A comparative study on automated software test oracle methods. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 140–145, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Donald R. Slutz. Massive stochastic testing of sql. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 618–622, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[66] Donald R. Slutz. Massive stochastic testing of sql. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 618–622, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[67] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 391–400, New York, NY, USA, 2011. ACM.

[68] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, November 1996.

[69] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mseqgen: object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 193–202, New York, NY, USA, 2009. ACM.

[70] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, September 2005.

[71] Bill Venners. ScalaTest. `http://www.scalatest.org`.

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

[73] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[74] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for java containers using state matching. In *Proceedings of the 2006 international symposium*

on Software testing and analysis, ISSTA '06, pages 37–48, New York, NY, USA, 2006. ACM.

[75] Yurong Wang. Test advising framework. Master's Thesis, Department of Computer Science and Engineering, University of Nebraska - Lincoln, January 2013.

[76] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 249–260, New York, NY, USA, 2008. ACM.

[77] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng.*, 6(3):236–246, May 1980.

[78] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 380–403, Berlin, Heidelberg, 2006. Springer-Verlag.

[79] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[80] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM.

[81] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[82] Sai Zhang, David Saff, Yingyi Bu, , and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proc. 11th International Symposium on Software Testing and Analysis (ISSTA 2011)*, 2011.

[83] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with mut-aware sequence recommendation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 293–296, New York, NY, USA, 2010. ACM.