Improving Test Suites via Generated Specifications

Technical Report MIT-LCS-TR-848

June 4, 2002

Michael Harder
MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
mharder@lcs.mit.edu

Abstract

This thesis presents a specification-based technique for generating, augmenting, and minimizing test suites. The technique is automatic but assumes the existence of a test case generator. The technique dynamically induces specifications from test suite executions. Test suites can be generated by adding cases until the induced specification stops changing. The resulting test suites have better fault detection than suites of the same size with 100% branch coverage. Augmenting an existing test suite, such as a code-covering suite, also increases its fault detection. Minimizing test suites while holding the generated specification constant compares favorably to previously-known techniques.

These positive results can be explained by two insights, which the thesis also justifies experimentally. First, given an a priori specification (an oracle), the *specification coverage* of a test suite compares the suite's induced specification with the oracle. Experiments show that specification coverage is correlated with fault detection, even when test suite size and code coverage are held constant. Second, when tests are added at random to a suite, specification coverage increases rapidly, then levels off at a high value. Even without knowing the ideal specification that would be induced by all possible tests, it is possible to produce a specification very near that one.

The thesis's test suite generation and augmentation technique increases the specification coverage of the test suite, but without knowing the oracle specification and without examining the code. In addition to improving fault detection, the technique generates a specification close to the oracle, which has many benefits in itself.

This technical report is a reformatted version of the author's Masters thesis of the same title, published by MIT in May, 2002. He was advised by Michael Ernst, Assistant Professor of Electrical Engineering and Computer Science.

Contents

Chapter 1

Introduction

Program specifications are useful in many aspects of program development, including design, coding, formal verification, optimization, and maintenance. Specifications serve these purposes largely by documenting program behavior; this documentation is used by humans and, when formalized, by other programs.

Specifications also play a valuable role in dynamic analyses such as software testing [GG75b, ROT89, CRS96, OL99]. Previous research on specification-based testing has required software testers to provide a specification. The research is limited by the fact that very few programs are formally specified — most lack even `assert` statements and comments — and the desire of software engineers to improve existing test suites or to use other tools for selecting test cases.

This thesis proposes and evaluates new specification-based testing techniques that do not require a specification to be provided a priori, that automatically provide users with a specification for the program under test, and that enable augmentation, minimization, or generation of test suites in cooperation with any existing technique for test case generation. The generated specification provides the benefits of an a priori specification, such as revealing removed functionality and checking test results, for future testing.

The key ideas behind the research are automatic dynamic generation of specifications from program executions, and comparison of specifications to one another. The remainder of this thesis provides additional details and evaluation, showing how these ideas can be used to improve test suites.

1.1 Motivation

This thesis investigates a new technique for improving the fault detection of test suites. An ideal test suite detects every possible fault in a program. Such a test suite can be created by mapping every possible input to the desired output. However, this is infeasible for even small programs. A program taking two 32-bit integer arguments requires over 18 million billion test cases.

In practice, test suites only cover a small fraction of the input space. Test suite size is constrained by two factors: execution time and creation effort. The execution time dictates how convenient it is to run a test suite, and how frequently it can be run. For example, if a test suite needs to run once per day, its execution time cannot exceed 24 hours. Test suites also require considerable effort to create — most testing requires humans to generate the expected output for each test case.

There is a tradeoff between test suite size and fault detection. Adding test cases increases size, but only some test cases increase fault detection. The tester's goal is to select the test cases with the highest probability of detecting faults.

However, it is impossible to directly measure the fault detection of a test suite in the field. We don't know what faults exist in our program right now, and we can't know what faults we may introduce in the future. Instead, we create test suites using techniques we believe are good predictors of fault detection. Previous techniques include human intuition, code coverage based techniques, and specification based techniques.

A common method for generating test suites is human intuition. A tester studies the program's source code or specification, and writes test cases he believes will do a good job of detecting faults in the program. Special attention may be paid to boundary values in the program or specification, and test cases for reported bugs may be added to a regression test suite. The tester's job is complete when he "feels" he has thoroughly tested the code.

Using this method alone is undesirable for many reasons. It is time-consuming, error-prone, and inconsistent. It is very easy for a tester to miss an important test case, and different testers may produce very different test suites.

A more formal technique for generating test suites uses code coverage metrics. The code coverage of a test suite measures how many source code constructs are exercised. Popular code coverage metrics include statement coverage and branch coverage. Statement coverage is defined as the number of program statements executed at least once, divided by the total number of reachable statements. Branch coverage is similar, except it counts branches taken instead of statements executed. To improve a test suite using a code coverage metric, test cases are added

until the metric reaches a certain level, usually 100%.

Code coverage techniques are well-known, well-studied, and have good tool support. However, they also have weaknesses. First, the technique does not guarantee 100% fault detection. In our experiments, test suites with 100% code coverage had fault detection rates varying from 16% to 96% (Section 4.4). Second, there is the problem of identifying which features in the program are reachable. Static analyses can identify some, but not all, unreachable program fragments. Third, it is unclear what the technique recommends for test suites that already have 100% code coverage. Possibilities include combining independent test suites with 100% coverage, or requiring that each code feature be covered multiple times.

The code coverage techniques are also displeasing philosophically. What we really care about is a program's *behavior*, not its source code. Measuring properties of the source code seems one step removed from our true concern. A more promising technique may be to measure properties of a program's behavior, as captured by a specification.

Several techniques have been proposed for using specifications in testing. Most of the techniques generate test cases, or test case specifications, from a program specification. Other techniques select test cases using a notion of *specification coverage* similar to code coverage; each property of the program specification must be covered at least once. Previous specification-based techniques require the programmer to provide an a priori program specification. This prevents the tools from being widely used, since most programs are not formally specified.

This thesis introduces and evaluates the specification difference (SD) technique. The SD technique attempts to address the weaknesses of the previous techniques. It is fully automatic, presuming the existence of a test case generator. It is specification based, but requires no a priori specification. All specifications are automatically generated from program executions. And it is complimentary to code coverage techniques, because it can improve test suites with 100% code coverage.

1.2 Approach

The specification difference technique (SD) is based on comparing specifications automatically generated from test suite executions. The technique relies on the following assumption. If two test suites produce different specifications, and one suite is a subset of the other, we assume the specification generated from the larger test suite is closer to the *oracle specification*. The oracle specification contains all true statements in the specification grammar.

We feel this is a valid assumption, based on the properties of dynamically analyses. Dynamically analyses are necessarily unsound, since they can only reason based on program executions they have seen thus far. Given more information, a dynamic analysis will generally produce a result no worse than its previous result.

Our current implementation of the SD technique uses the Daikon invariant detector [ECGN01, Ern00] to dynamically generate specifications from execution traces, but a different dynamic specification generator could easily be used instead. Daikon generates a likely specification by observing program executions. It reports invariants that were true for all executions and deemed unlikely to happen by chance.

The specification difference technique generates test suites by adding test cases until the specification stops changing. At this point, the generated specification will be close to the oracle specification (Section 6.3). It turns out that these test suites have good fault detection (Chapter 4). This makes intuitive sense — if the program specification can be recovered from the test suite, the suite must be exercising the program well.

1.3 Contributions

The thesis of this research is that specifications generated from program executions can be leveraged for testing tasks.

The first contribution of this research is the proposal of the specification difference (SD) technique for generating, augmenting, and minimizing test suites. The technique is fully automatic (assuming the existence of a test case generator), and no a priori specification is required. This makes the technique more likely to be used in practice, since the barrier to entry is quite low. The SD technique can be combined with code coverage techniques by augmenting an existing suite with 100% code coverage. As an additional benefit, the SD technique generates an accurate specification for the program under test.

The second contribution of this research is the experimental evaluation of the SD technique. We compare the average size and fault detection of suites generated at random, by the SD technique, and by code coverage techniques. The SD technique produces suites of comparable size and fault detection to those produced by branch coverage techniques. We also determine what types of faults are best detected by the SD technique. The SD technique is better than the branch coverage technique at detecting faults that do not involve a change to the control-flow graph of the program. SD-augmenting a suite with 100% branch coverage yields test suites with better fault detection than either technique alone.

The third contribution of this research is a new notion of specification coverage. A related contribution is a technique for determining whether two formal specifications are different, and how much they differ. Prior notions of specification coverage assumed a specification clause was covered if it was satisfied at least once [CR99]. Our specification generator uses statistical tests to determine the most likely specification. Thus, it may require several test

cases to cover a clause in the specification. We believe a statistical method gives a more accurate evaluation of test suite quality. If a specification asserts that two variables are equal, we should not consider the assertion covered until we have seen many executions where the assertion is true.

1.4 Outline

The remainder of this work is organized as follows.

Chapter 2 provides the relevant background for this work. It defines what we mean by "specification" and how we generate specifications from program executions.

Chapter 3 describes the specification difference technique in more detail, and presents an example showing how a test suite is generated.

Chapter 4 experimentally evaluates the specification difference (SD) technique. The results show that the SD technique compares favorably to previous code coverage based techniques.

Chapter 5 describes and evaluates enhancements we made to improve the SD technique.

Chapter 6 describes our notion of *specification coverage* and explains how we compare specifications to one another. It presents insights that justify the results of Chapter 4 — specification coverage is correlated with fault detection, and if it is poor, it increases rapidly as cases are added to a test suite.

Chapter 7 summarizes related work. Previous research in specification-based testing has primarily focused on test case generation, and has required an a priori specification.

Chapter 8 outlines future work, which contains ideas about the specification difference technique and additional applications of generated specifications.

Chapter 9 concludes.

Chapter 2

Background

This chapter first provides some background about specifications. Then it briefly overviews our dynamic (runtime) technique for automatically inducing specifications from program executions — equivalently, from a program plus a test suite. The specific details are not germane to the present thesis, which uses the specification generator as a black box, and which could equally well use any other technique for generating specifications from a test suite. Last, this chapter explains what we mean by "specification", since it is a rather vague term in software engineering.

2.1 Formal specifications

A formal specification is a precise mathematical abstraction of program behavior [LG01, Som96, Pre92]. Specifications often state properties of data structures, such as object invariants, or relate variable values in the pre-state (before a procedure call) to their post-state values (after a procedure call). In this thesis, we focus on code specifications rather than high-level specifications that assert properties about an abstraction represented by program structures. Some definitions require that the specification be an a priori description of intended or desired behavior that is used in prescribed ways; we do not adopt that definition here.

A specification for a procedure that records its maximum argument in variable *max* might include

$$\text{if } arg > max \text{ then } max' = arg \text{ else } max' = max$$

where *max* represents the value at the time the procedure is invoked and max' represents the value of the variable when the procedure returns. A typical specification contains many clauses, some of them simple mathematical statements and others involving post-state values or implications. The clauses are conjoined to produce the full specification. These specification clauses are often called *invariants*. There is no single best specification for a program; different specifications include more or fewer clauses and assist in different tasks. Likewise, there is no single correct specification for a program; correctness must be measured relative to some standard, such as the designer's intent, or task, such as program verification.

2.2 Automatic dynamic generation of specifications

Our technique for inducing specifications uses dynamic invariant detection [ECGN01, Ern00] to extract likely program invariants from program executions, then combines the reported invariants into a specification. An invariant is a property that is true at some point or points in the program, such as a method precondition or postcondition, an object invariant, or the condition of an `assert` statement. Our experiments use the Daikon invariant detector, whose output includes conditionals, pre- and post-state variables, and properties over a variety of data structures [EGKN99]. Thus, the output is often a high-quality specification that matches human-written formal specifications or can be proved correct [NE01b, NE01a]. Even lesser-quality output forms a partial specification that is nonetheless useful [KEGN01]. The specifications are unsound: the properties are likely, but not guaranteed, to hold.

Full details on dynamic invariant detection appear elsewhere [ECGN01, Ern00], but those details are not relevant to the present thesis. Briefly, a dynamic invariant detector discovers likely invariants from program executions by running the program, examining the values that it computes, and detecting patterns and relationships among those values. The detection step uses an efficient generate-and-check algorithm that reports properties that hold over execution of an entire test suite (which is provided by the user). The output is improved by suppressing invariants that are not statistically justified and by other techniques [ECGN00]. (The statistical tests use a user-settable confidence parameter. The results in this thesis use the default value, which is .01. We repeated the experiments with values between .0001 and .99, and the differences were negligible. Therefore, we conclude that our results are not sensitive to that parameter.)

As with other dynamic approaches such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test suite. When a reported invariant is not universally true for all possible executions, then it indicates a property of the program's context or environment or a deficiency of the

test suite. In many cases, a human or an automated tool can examine the output and enhance the test suite.

The Daikon invariant detector is language independent, and currently includes instrumenters for the C [KR88], IOA [GL00], and Java [AGH00] languages. Daikon is available from `http://pag.lcs.mit.edu/daikon/`. While our experiments rely on the Daikon tool, the ideas generalize beyond any particular implementation of specification generation. Our experiments used a version of Daikon that instruments source code, but both a previous and a more recent version of Daikon work directly on binaries. The dynamic invariant detection technique does not require source and does not depend on any internal structure of the program component being analyzed; in that sense, the technique is black-box.

2.3 Types of specifications

Specifications are used in many different stages of development, from requirements engineering to maintenance. Furthermore, specifications take on a variety of forms, from a verbal description of customer requirements to a set of test cases or an executable prototype. In fact, there is no consensus regarding the definition of "specification" [Lam88, GJM91].

Our research uses formal specifications, as defined in Section 2.1. This definition is standard, but our *use* of specifications is novel. Our specifications are generated automatically, after an executable implementation exists (Section 2.2). Typically, software engineers are directed to write specifications before implementation, then to use them as implementation guides — or simply to obtain the benefit of having analyzed requirements at an early design stage [Som96].

Despite the benefits of having a specification before implementation, in practice few programmers write (formal or informal) specifications before coding. Nonetheless, it is useful to produce such documentation after the fact [PC86]. Obtaining a specification at any point during the development cycle is better than never having a specification at all. *Post hoc* specifications are also used in other fields of engineering. As one example, speed binning is a process whereby, after fabrication, microprocessors are tested to determine how fast they can run [Sem94]. Chips from a single batch may be sold with a variety of specified clock speeds.

Some authors define a specification as an a priori description of intended or desired behavior that is used in prescribed ways [Lam88, GJM91]. For our purposes, it is not useful to categorize whether a particular logical formula is a specification based on who wrote it, when, and in what mental state. (The latter is unknowable in any event.) Readers who prefer the alternative definition may replace the term "specification" by "description of program behavior" (and "invariant" by "program property") in the text of this thesis.

We believe that there is great promise in extending specifications beyond their traditional genesis as pre-implementation expressions of requirements. One of the contributions of our research is the insight that this is both possible and desirable, along with evidence to back up this claim.

Chapter 3

Technique

This chapter describes the specification difference (SD) technique for generating, augmenting, and minimizing test suites. The SD technique compares the specifications induced by different test suites in order to decide which test suite is superior. (Sections 6.3 and 6.4 justify the technique.)

The specification difference technique is fully automatic, but for test suite generation and augmentation, it assumes the existence of a test case generator that provides candidate test cases. In other words, our technique selects, but does not generate, test cases. Test cases may be generated at random or from a grammar, created by a human, selected from a test pool, or produced in any other fashion.

3.1 Generation

This section describes the most basic specification difference generation technique, which will be referred to as SD-base for the remainder of this thesis. Start with an empty test suite, which generates an empty specification. Generate a new specification from the test suite augmented by a candidate test case. If the new specification differs from the previous specification, add the candidate test case to the suite. Repeat this process until some number n of candidate cases have been consecutively considered and rejected. A pseudocode implementation of the technique is presented in Figure 3.1.

This is similar to a previous technique for generating suites with 100% branch coverage [RH98]. Starting with an empty test suite, add candidate test cases as long as they increase the branch coverage of the suite. Repeat until the suite contains at least one case that would exercise each executable branch in the program.

There are two key differences between the SD-base technique and the branch coverage technique. First, adding a test case can only improve the branch coverage of a suite, but it may either improve or worsen the generated specification of a suite. We don't know the goal specification, so we don't know if a changed specification is closer to or farther from the goal. Second, the branch coverage technique is finished when the suite covers all executable branches. However, the SD-base technique is finished when the specification stops changing. Again, we don't know the goal specification, so we can never know with certainty when to stop adding test cases. This issue is discussed in Section 8.1.5.

```
procedure SD-Generate (program P, int n)
  testsuite T ← {}
  int i ← 0
  while i < n do
    testcase c ← Newcase()
    if Spec(P, T) ≠ Spec(P, T ∪ {c}) then
      T ← T ∪ {c}
      i ← 0
    else
      i ← i + 1
    end if
  end while
  return T
```

Figure 3.1: Pseudocode implementation of the SD-base test suite generation technique. Newcase is a user-specified procedure that generates a candidate test case; our experiments randomly select test cases from a pool. Spec is a procedure that generates a specification from a program and a test suite; our experiments use the Daikon invariant detector (Section 2.2).

3.1.1 Enhancements

We made two enhancements to the SD-base technique to improve the quality of the generated test suites. First, we selected candidate test cases using the choose4 algorithm (Section 5.1), which evaluates four candidate cases simultaneously and selects the case that changes the specification most. Second, after generating a test suite we minimized it using the SD minimization technique (Section 3.3). For the remainder of this thesis, when we refer to the SD generation technique, we mean the SD-base technique plus these two enhancements.

In Chapter 5, we explain the choose4 algorithm, present evidence showing that these enhancements improve the

```
procedure SD-Minimize (program P, testsuite T)
  specification S = Spec(P, T)
  for all testcase c ∈ T do
    if Spec(P, T − {c}) = S then
      T ← T − {c}
    end if
  end for
  return T
```

Figure 3.2: Pseudocode implementation of the specification difference test suite minimization technique. Spec is a procedure that generates a specification from a program and a test suite; our experiments use the Daikon invariant detector (Section 2.2).

SD-base generation technique, and provide intuition for why the enhancements work.

3.2 Augmentation

The specification difference augmentation technique is identical to the SD-base generation technique, except the process is started with an existing test suite rather than an empty test suite. We did not make any enhancements to the SD augmentation technique, although we plan to evaluate the choose4 enhancement in the future.

3.3 Minimization

The specification difference minimization technique operates analogously to the SD generation and augmentation techniques. Starting with a test suite, generate a specification. For each test case in the suite, consider the specification generated by the test suite with that case removed. If the specification does not differ, then remove the test case from the suite. The resulting suite generates the same specification as the original, but is smaller. A pseudocode implementation of the technique is presented in Figure 3.2.

As with many minimization techniques, the SD technique does not guarantee that the minimized suite is the absolute minimum set of cases that can generate the original specification. To do so could have a worst-case runtime of 2^s, where s is the size of the original suite, since it could require examining every subsuite of the original suite. (A technique like Delta Debugging [ZH02] may be able to reduce the cost.) Generated specifications are a multiple-entity criterion in that no one test case alone guarantees that a particular invariant is reported; rather, several cases may be required, and adding cases may even reduce the size of the generated specification.

3.4 Example

This section presents an example of generating a test suite using the SD-base technique (Section 3.1). Assume we are using $n = 3$, meaning the generation process will terminate once 3 test cases have been consecutively considered and rejected. Also assume we have a specification generator that reports the following properties at function entry and exit points:

- $var = constant$
- $var \geq constant$
- $var \leq constant$
- $var = var$
- $property \Rightarrow property$

The last item is a way to express conditional properties — when the first property is true, the second property must also be true.

We will apply the SD-base technique to a function that computes the absolute value of an integer:

```
int abs(int x) {
  if (x >= 0)
    return x;
  else
    return -x;
}
```

We start with an empty test suite and empty specification. Next, we select a candidate test case from our test case generator. We select the test case "5", so we generate a specification from our program and the test suite {5}:

```
abs:ENTER
  x == 5
abs:EXIT
  x == orig(x)
  return == x
```

The first property states that `x` equals 5 at the function entrance. The second property states that the value of `x` at the procedure exit equals the value of `x` at the procedure entry. The third property states that the return value of the function is equal to `x`. This isn't a very good specification of the absolute value function, but it is accurate for the one test case we have seen.

The specification generated for the test suite {5} is different from the specification for the empty test suite, so the test case "5" is accepted. The results of adding subsequent test cases are summarized in Figure 3.3.

The process is terminated after test case "4", since 3 test cases have been consecutively considered and rejected. The final test suite generated by the SD-base technique is $\{5, 1, -1, -6, 0\}$.

Test case	Specification (if different from previous)	
	`abs:ENTER`	`abs:EXIT`
5	`x == 5`	`x == orig(x)` `x == return`
1	`x >= 1`	`x == orig(x)` `x == return`
4		

−1		`(x >= 1) => (x == return)`
		`(x == -1) => (x == -return)`
		`return >= 1`

−6		`(x >= 1) => (x == return)`
		`(x <= -1) => (x == -return)`
		`return >= 1`

−3		

0		`(x >= 0) => (x == return)`
		`(x <= -1) => (x == -return)`
		`return >= 0`

7		
−8		
4		

Figure 3.3: Example of generating a test suite via the SD-base technique. The test cases are considered in order from top to bottom, and added to the suite if the specification changes. The process terminates when 3 test cases have been consecutively considered and rejected.

Chapter 4

Evaluation

This chapter evaluates the SD generation, augmentation, and minimization techniques by comparing them to previous code coverage based techniques.

4.1 Subject programs

Our experiments analyze eight C programs. Each program comes with a pool of test cases and faulty versions. Figure 4.1 lists the subjects.

The first seven programs in Figure 4.1 were created by Siemens Research [HFGO94], and subsequently modified by Rothermel and Harrold [RH98]. The Siemens researchers generated tests automatically from test specification scripts, then augmented those with manually-constructed white-box tests such that each feasible statement, branch, and def-use pair was covered by at least 30 test cases. The Siemens researchers created faulty versions of the programs by introducing errors they considered realistic. Each faulty version differs from the canonical version by 1 to 5 lines of code. They discarded faulty versions that were detected by more than 350 or fewer than 3 test cases; they considered the discarded faults too easy or too hard to detect. (A test suite detects a fault if the output of the faulty and correct versions differ.)

The `space` program interprets Array Definition Language inputs and has been used as a subject in a number of testing studies. The test pool for `space` contains 13585 cases. 10000 were generated randomly by Vokolos and Frankl [VF98], and the remainder were added by Graves et al. [GHK+01] until every edge was covered by at least 30 cases. Each time an error was detected during the program's development or later by Graves et al., a new faulty version of the program (containing only that error) was created.

Some of our experiments use test suites randomly generated by selecting cases from the test pool. Other experiments use statement, branch, and def-use coverage suites generated by Rothermel and Harrold [RH98]. These suites were generated by picking tests from the pool at random and adding them to the suite if they added any coverage, until all the coverage conditions were satisfied. There are 1000 test suites for each type of coverage, except there are no statement or def-use covering suites for `space`.

For the SD generation and augmentation techniques, we generated candidate test cases by selecting randomly from the test pool.

4.2 Measurement details

This section describes how various measurements were performed. Others, such as code size, are standard.

Test suite size. We measured test suite size in terms of test cases and function calls. Our primary motivation for measuring these quantities is to control for them to avoid conflating them with other effects.

Each test case is an invocation of the program under test. This measure of size is most readily apparent to the tester: in our case, it is the number of lines in the script that runs the suite.

The number of test cases is an incomplete measure of test suite size, because a single case might execute only a few machine instructions or might run for hours. Therefore, we also measured the number of non-library function calls performed during execution of the test suite. This is a more accurate measure of how much the test suite exercises the program, and it is an approximation of the runtime required to execute the test suite.

Code coverage. For simplicity of presentation, we use the term "code coverage" for any structural code coverage criterion. We measured statement coverage using the GCC `gcov` tool and branch coverage using Bullseye Testing Technology's C-Cover tool. Unreachable code and unrealizable paths in the programs prevent the tools from reporting 100% coverage. We normalized all code coverage measurements by dividing by the fraction of statements or branches covered by the pool of test cases. The fraction of reachable statements ranged from 93% to 98%, and the fraction of reachable branches ranged from 87% to 97%.

Fault Detection. A test suite detects a fault (actually, detects a faulty version of a program) if the output of the faulty version differs from the output of the correct version, when both are run over the test suite. The fault

Program	Program size			Faulty	Test pool			Description
	Functions	LOC	NCNB	versions	cases	calls	spec. size	of program
print_tokens	18	703	452	7	4130	619424	97	lexical analyzer
print_tokens2	19	549	379	10	4115	723937	173	lexical analyzer
replace	21	506	456	30	5542	1149891	252	pattern replacement
schedule	18	394	276	9	2650	442179	283	priority scheduler
schedule2	16	369	280	9	2710	954468	161	priority scheduler
tcas	9	178	136	41	1608	12613	328	altitude separation
tot_info	7	556	334	23	1052	13208	156	information measure
space	136	9568	6201	34	13585	8714958	2144	ADL interpreter

Figure 4.1: Subject programs used in experiments. "LOC" is the total lines of code; "NCNB" is the number of non-comment, non-blank lines of code. Test suite size is measured in number of test cases (invocations of the subject program) and number of non-library function calls at runtime. Specification size is the number of invariants generated by the Daikon invariant detector when run over the program and the full test pool.

detection rate of a test suite is the ratio of the number of faulty versions detected to the number of faulty program versions. Section 4.1 describes how faulty versions were selected.

4.3 Comparing techniques

This section describes how we compared the SD technique to code coverage techniques. The techniques generate test suites of different sizes, and thus it is impossible to directly compare them. Instead, we create *stacked suites* by combining suites of one technique until they reach the size of the other technique. Since the sizes are equal, we can compare the techniques using just their fault detection.

Random selection can create a test suite of any size — the desired size is an input to the random selection process. In contrast, the SD and code-coverage techniques produce test suites of a specific size. For a certain program and test case generator, the SD technique may produce suites averaging 20 cases, while the branch coverage technique may produce suites averaging 10 cases. We call this the *natural size* of a technique for a given program and test case generator. There is no direct way to use the technique to produce test suites with different average size.

We are primarily interested in two properties of a test suite improvement technique: the average size and average fault detection of test suites created by the technique (Section 1.1). Smaller size is better, and higher fault detection is better. When comparing two techniques, there are three combinations of these two variables. Consider the four techniques shown in Figure 4.2.

Technique A is better than B because it produces suites that are smaller but have the same fault detection. Techniques C is better than B because it produces suites with the same size but higher fault detection. However, it isn't possible to directly compare A to C. Technique A produces smaller suites, but C produces suites with higher fault detection. Either technique may be better depending on one's needs.

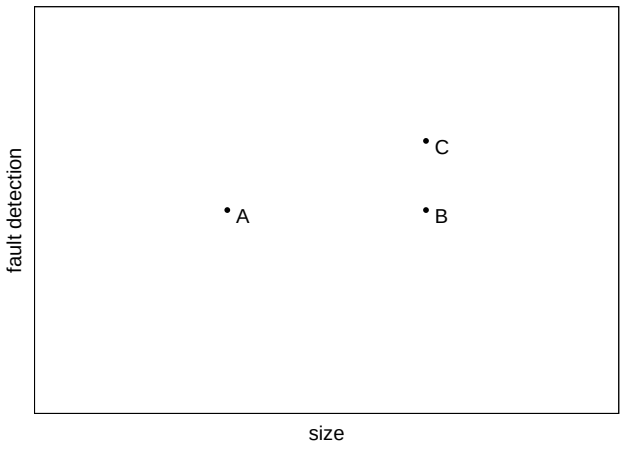

Figure 4.2: Average size and average fault detection of test suites generated by three different techniques. Technique A is better than B, and C is better than B, but A and C cannot be directly compared because they differ in both size and fault detection.

One way to compare A to C is to determine which would have higher fault detection *if* the suites were the same size. Assume technique A could be used to generate suites of any size. Then point A in Figure 4.2 would be just one point on the size to fault detection curve of technique A. If we could construct this curve, we could determine whether point C lies above, below, or on the curve, as shown in Figure 4.3.

If point C lies above the curve, then technique C is better than technique A, since technique C produces suites with higher fault detection at its natural size point. Similarly, if point C lies below the curve, then technique C is worse than technique A. Finally, if point C lies on the curve, the techniques are about equal. We could perform a similar comparison by examining the size to fault detection curve of technique C and point A.

This leaves the question of how to generate the size to fault detection curves for the SD and code coverage

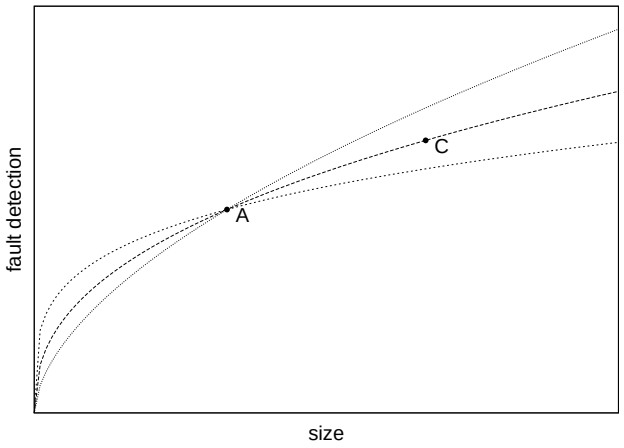

Figure 4.3: Different potential size vs. fault detection curves for technique A.

techniques. We use a simple algorithm we call *stacking*. Assume we have a pool of test suites generated by a technique. To generate a stacked suite of size s, we first select a random suite from the pool. We add its test cases to the stacked suite, then select another suite and add its cases. This is repeated until the stacked suite reaches size s. To reach exactly size s, it is likely we will add only part of the last suite selected from the pool. We select the cases randomly to avoid any bias in the natural ordering of the cases within the suite. The stacking algorithm can generate test suites of any size from any technique.

To compare techniques A and C in our example, generate suites using the stacked-A technique with size equal to the suites of technique C. If the suites of technique C have higher fault detection, then C is the better technique. If the suites of technique C have lower fault detection, then A is the better technique. Similarly, one could generate suites using the stacked-C technique with size equal to the suites of technique A.

These two comparisons may disagree: technique A could be better than stacked-C, while C could be better than stacked-A. In this case, the stacking algorithm would be unable to determine which technique is better.

We use the stacking algorithm to compare the SD and branch coverage techniques in Sections 4.4 and 4.5, and to evaluate enhancements to the SD-base technique in Chapter 5.

Instead of stacking, one could compare test suites by their so-called *efficiency*, or fault detection to size ratio. However, we believe this measurement is not meaningful, because its judgments may contradict our more reasonable ones.

4.4 Generation

To evaluate the SD generation technique, we automatically generated test suites for each of the subject programs. We compared these to other automatic generation techniques, including random selection, statement coverage, branch coverage, and def-use coverage.

We used $n = 50$ when generating test suites, meaning the process is terminated when 50 consecutive test cases have been considered without changing the specification. The value chosen for n is a tradeoff between the running time of the generation process and the quality of the generated suite. For our subject programs, $n = 50$ gave a high level of fault detection and moderate generation time.

For each subject program, we generated 50 test suites using the SD technique and measured their average size and fault detection. Figure 4.4 compares them to other automatically generated suites. First, we compare SD suites to code coverage suites (statement, branch, and def-use) at their natural sizes (Section 4.3). The SD suites have better fault detection, and are larger, than the statement and branch covering suites. The SD suites have worse fault detection, but are smaller, than the def-use covering suites.

The SD suites are closest in size and fault detection to the branch covering suites, so we created suites by stacking (Section 4.3) the branch covering suites to equal the size of the SD suites, and vice-versa. We also created random suites to equal the size of the SD and branch covering suites.

The SD and branch coverage suites have much higher fault detection than randomly generated suites of the same size. In other words, both the SD and branch coverage techniques capture some important aspect related to fault detection.

When the SD and branch coverage techniques are compared via stacking, the result is inconclusive. At the natural size of SD suites, the SD technique performs 8% better than branch coverage. However, at the natural size of branch coverage suites, the SD technique performs 1% worse than branch coverage.

4.4.1 Detecting specific faults

Depending on the desired size of the test suites, the SD technique performs from somewhat better to slightly worse than branch coverage. This may not provide a compelling reason to use the SD technique. However, Figure 4.4 shows that the SD technique is superior for detecting faults in certain programs, and it may be better at detecting certain types of faults.

We compared the individual fault detection rates for the SD technique and the stacked branch coverage technique. For each fault in each program, we measured the proportion of times the fault was detected by each technique. We then used a nonparametric $P1 = P2$ test to determine if there was a statistically significant difference between the two proportions, at the $p = .05$ level.

For example, fault 7 in `print_tokens` was detected 14

	SD		Statement		Branch		Def-use	
	fault	size	fault	size	fault	size	fault	size
print_tokens	.366	9.3	.409	14.6	.457	16.1	.774	38.6
print_tokens2	.506	6.4	.778	11.7	.738	12.0	.966	35.0
replace	.451	18.1	.347	18.5	.361	18.7	.787	64.9
schedule	.329	10.2	.209	5.8	.442	8.6	.762	23.9
schedule2	.304	13.1	.209	6.9	.242	7.7	.522	25.6
tcas	.547	25.6	.180	5.2	.176	5.8	.163	5.5
tot_info	.719	9.4	.530	6.8	.560	7.4	.704	15.0
Average	.460	13.2	.380	9.9	.425	10.9	.670	29.8
space	.795	62.5	*	*	.909	155.2	*	*
Grand Average	.502	19.3	*	*	.486	28.9	*	*

	SD		Stack Branch		Random	
	fault	size	fault	size	fault	size
print_tokens	.366	9.3	.329	9.3	.151	9.3
print_tokens2	.506	6.4	.446	6.4	.378	6.4
replace	.451	18.1	.357	18.1	.264	18.1
schedule	.329	10.2	.449	10.2	.251	10.2
schedule2	.304	13.1	.367	13.1	.153	13.1
tcas	.547	25.6	.422	25.6	.400	25.6
tot_info	.719	9.4	.553	9.4	.420	9.4
space	.795	62.5	.778	62.5	.701	62.5
Average	.502	19.3	.463	19.3	.340	19.3

	Branch		Stack SD		Random	
	fault	size	fault	size	fault	size
print_tokens	.457	16.1	.457	16.1	.186	16.1
print_tokens2	.738	12.0	.658	12.0	.466	12.0
replace	.361	18.7	.458	18.7	.285	18.7
schedule	.442	8.6	.322	8.6	.209	8.6
schedule2	.242	7.7	.191	7.7	.078	7.7
tcas	.176	5.8	.214	5.8	.116	5.8
tot_info	.560	7.4	.665	7.4	.424	7.4
space	.909	155.2	.884	155.2	.811	155.2
Average	.486	28.9	.481	28.9	.322	28.9

Figure 4.4: Test suites created via automatic generation techniques. Fault is the fraction of faults detected. Size is the number of test cases in the suite. All numbers are averaged across 50 suites of each type for each program. Statement and def-use coverage suites were not available for `space`.

out of 50 times by the SD technique, and 6 out of 50 times by the stacked branch coverage technique. According to the statistical test, these proportions are significantly different. We conclude that the SD technique is better than the stacked branch coverage technique for detecting this fault.

There are a total of 163 faults in the 8 programs we examined. The SD technique is better at detecting 65 faults, stacked branch coverage is better at detecting 33 faults, and the difference is not statistically significant for 65 faults.

We examined each fault by hand, to determine if there was a qualitative difference between the faults detected best by the SD technique, and the faults detected best by the stacked branch coverage technique. We determined whether each fault changed the control flow graph (CFG) of the program. We treated basic blocks as nodes of the CFG, so adding a statement to a basic block would not change the CFG. Examples of CFG changes include: adding or removing an `if` statement, adding or removing a case from a `switch` statement, and adding or removing a `return` statement. Examples of non-CFG changes include: adding or removing a statement from a basic block, changing a variable name, changing an operator, and modifying the expression in the condition of an `if` statement. If a fault is not a CFG change, it must be a change to the value of an expression in the program. Our results are presented in the following table.

	SD Better	Same	Branch Better	Total
CFG	9	11	9	29
Non-CFG	56	54	24	134
Total	65	65	33	163

For detecting CFG changes, the SD technique performs as well as stacked branch coverage. For detecting non-CFG changes (changes to values), the SD technique performs over 2 times better than stacked branch coverage. This makes intuitive sense, because our specifications contain assertions about the values of variables at different points in the program. Finally, the value changes outnumber the CFG changes by a factor of 4. These faults were created by people who considered them realistic, so we can assume the distribution of faults will be similar in

	Branch		SD Aug		Stack Branch		Stack SD	
	fault	size	fault	size	fault	size	fault	size
print_tokens	.457	16.1	.514	19.1	.446	19.1	.411	19.1
print_tokens2	.738	12.0	.762	14.8	.764	14.8	.588	14.8
replace	.361	18.7	.559	30.5	.471	30.5	.534	30.5
schedule	.442	8.6	.553	18.3	.629	18.3	.436	18.3
schedule2	.242	7.7	.396	22.9	.460	22.9	.389	22.9
tcas	.176	5.8	.645	49.2	.538	49.2	.638	49.2
tot_info	.560	7.4	.748	15.6	.691	15.6	.730	15.6
space	.909	155.2	.921	161.6	.908	161.6	.876	161.6
Average	.486	28.9	.637	41.5	.613	41.5	.575	41.5

Figure 4.5: Test suites created using the SD technique to augment branch coverage suites. Fault is the fraction of faults detected. Size is the number of test cases in the suite. All numbers are averaged across 50 suites of each type for each program.

	Orig		SD		Random		Branch-Min	
	fault	size	fault	size	fault	size	fault	size
print_tokens	.651	100	.549	48.2	.443	48.2	.314	7.3
print_tokens2	.920	100	.616	14.4	.540	14.4	.740	6.1
replace	.657	100	.443	19.8	.281	19.8	.289	10.5
schedule	.647	100	.449	30.6	.391	30.6	.240	4.8
schedule2	.649	100	.451	39.9	.331	39.9	.231	4.8
tcas	.709	100	.505	26.2	.417	26.2	.177	4.9
tot_info	.887	100	.683	18.0	.539	18.0	.501	5.2
space	.754	100	.736	59.4	.685	59.4	.740	48.0
Average	.734	100	.554	32.1	.453	32.1	.404	11.5

Figure 4.6: Test suites minimized via automatic techniques. Fault is the fraction of faults detected. Size is the number of test cases in the suite. All numbers are averaged across 50 suites of each type for each program.

practice.

4.5 Augmentation

To evaluate the SD augmentation technique, we automatically augmented 50 branch coverage suites for each of the programs. Figure 4.5 compares these to the original suites, to suites augmented by stacking branch coverage suites, and to suites of the same size created by stacking SD-generated suites. As with generation, we used $n = 50$.

On average, suites augmented using the SD technique have 4% higher fault detection than stacked branch suites of the same size. Furthermore, the SD-augmented suites have 11% higher fault detection than stacked SD suites of the same size. There are two conclusions to draw from this data. First, the SD technique can be used to improve the fault detection of suites with 100% branch coverage. Second, combining the SD and branch coverage techniques yields test suites with higher fault detection than either technique alone.

4.6 Minimization

For each program, we generated 50 random test suites with 100 cases each. We minimized these by the SD minimization technique, by random sampling, and by maintaining branch coverage. Figure 4.6 shows the results.

Suites minimized by the SD technique are much smaller than the original randomly generated suite, but also have less fault detection. The SD technique has better fault detection than random minimization. It results in better fault detection, but also larger test suites, than minimizing while maintaining branch coverage.

Chapter 5

Enhancements

This chapter describes and evaluates several enhancements to the basic specification difference generation technique (Section 3.1).

5.1 Choose4

Choose4 is an alternate algorithm for selecting test cases. In the SD-base technique, candidate test cases are considered one at a time. If the test case causes the specification to change, the test case is added. In the SD-choose4 technique, four candidate test cases are considered simultaneously. The test case that causes the specification to change *most* is added, and the other three cases are discarded. If none of the cases cause the specification to change, all of the cases are discarded. The SD-base and SD-choose4 techniques are otherwise identical. The SD-choose4 technique is terminated when n consecutive test cases fail to change the spec (as opposed to n consecutive iterations of 4 test cases each).

The intuition behind the choose4 algorithm is as follows. The candidate test case that changes the specification most is somehow the most "different" from the test cases already in the suite. Since the candidate is the most different, it should be the most likely to detect a fault not already detected by the suite.

We performed an experiment to determine the efficacy of this enhancement. For each subject program, we generated 50 test suites using the SD-choose4 and SD-base techniques. To compare the two techniques, we stacked SD-base suites to equal the size of SD-choose4 suites, and vice-versa (Section 4.3). The results are presented in Figure 5.1. Size for size, the SD-choose4 technique generates test suites with higher fault detection than the SD-base technique.

5.2 SD minimization

The SD minimization technique (Section 3.3) can be applied to suites generated by the SD-base technique. We believe this will improve the SD-base technique, since it will remove the test cases not necessary to generate the specification. The intuition is that these test cases are similar to other cases in the suite, and thus will add little to fault detection. In particular, the cases added at the beginning of the SD-base generation technique are essentially chosen at random, since any test case will change the specification early on.

We performed an experiment to determine the efficacy of this enhancement. For each subject program, we generated 50 test suites using the SD-base technique, then applied the SD minimization technique. To compare the two techniques, we stacked SD-base suites to equal the size of SD-minimized suites, and vice-versa. The results are presented in Figure 5.2. Size for size, the SD-minimized technique generates test suites with higher fault detection than the SD-base technique.

5.3 Choose4 with SD minimization

The choose4 and SD minimization enhancements can both be applied to the SD-base technique. This is referred to as simply the SD technique (Section 3.1.1). For each subject program, we generated 50 test suites using the SD, SD-choose4, and SD-minimized techniques. To compare the three techniques, we stacked the SD-choose4 and SD-minimized techniques to equal the size of the SD technique, and vice-versa. The results are presented in Figure 5.3. Size for size, the SD technique generates suites with the highest fault detection.

	SD-choose4		Stacked SD-base		SD-base		Stacked SD-choose4	
	fault	size	fault	size	fault	size	fault	size
print_tokens	.369	11.6	.308	11.6	.394	15.6	.431	15.6
print_tokens2	.526	8.1	.436	8.1	.558	11.6	.604	11.6
replace	.513	25.5	.472	25.5	.517	29.7	.563	29.7
schedule	.384	14.2	.351	14.2	.387	17.1	.404	17.1
schedule2	.411	23.3	.382	23.3	.418	23.8	.444	23.8
tcas	.638	43.7	.610	43.7	.644	49.2	.651	49.2
tot_info	.732	11.8	.680	11.8	.739	14.2	.735	14.2
space	.814	79.1	.800	79.1	.821	91.4	.821	91.4
Average	.548	27.2	.505	27.2	.560	31.6	.582	31.6

Figure 5.1: Test suites created by the SD-choose4 and SD-base generation techniques. Fault is the fraction of faults detected. Size is the number of test cases in the suite. All numbers are averaged across 50 suites of each type for each program.

	SD-minimized		Stacked SD-base		SD-base		Stacked SD-minimized	
	fault	size	fault	size	fault	size	fault	size
print_tokens	.377	11.3	.294	11.3	.394	11.6	.414	11.6
print_tokens2	.528	9.7	.486	9.7	.558	8.1	.574	8.1
replace	.446	18.5	.387	18.5	.517	25.5	.530	25.5
schedule	.340	11.9	.331	11.9	.387	14.2	.404	14.2
schedule2	.327	13.6	.307	13.6	.418	23.3	.422	23.3
tcas	.539	26.0	.470	26.0	.644	43.7	.651	43.7
tot_info	.715	10.7	.649	10.7	.739	11.8	.719	11.8
space	.795	64.9	.761	64.9	.821	79.1	.825	79.1
Average	.508	20.8	.461	20.8	.560	27.2	.567	27.2

Figure 5.2: Test suites created by the SD-minimized and SD-base generation techniques. Fault is the fraction of faults detected. Size is the number of test cases in the suite. All numbers are averaged across 50 suites of each type for each program.

	SD		Stacked SD-ch4		Stacked SD-min		SD-ch4		Stacked SD		SD-min		Stacked SD	
	fault	size	fault	size	fault	size	fault	size	fault	size	fault	size	fault	size
print_tokens	.366	9.3	.309	9.3	.337	9.3	.369	11.6	.369	11.6	.377	11.3	.406	11.3
print_tokens2	.506	6.4	.404	6.4	.382	6.4	.526	8.1	.574	8.1	.528	9.7	.604	9.7
replace	.451	18.1	.440	18.1	.439	18.1	.513	25.5	.543	25.5	.446	18.5	.459	18.5
schedule	.329	10.2	.336	10.2	.320	10.2	.384	14.2	.391	14.2	.340	11.9	.320	11.9
schedule2	.304	13.1	.238	13.1	.307	13.1	.411	23.3	.411	23.3	.327	13.6	.262	13.6
tcas	.547	25.6	.520	25.6	.547	25.6	.638	43.7	.644	43.7	.539	26.0	.543	26.0
tot_info	.719	9.4	.634	9.4	.661	9.4	.732	11.8	.722	11.8	.715	10.7	.710	10.7
space	.795	62.5	.778	62.5	.785	62.5	.814	79.1	.823	79.1	.795	64.9	.798	64.9
Average	.502	19.3	.457	19.3	.472	19.3	.548	27.2	.560	27.2	.508	20.8	.513	20.8

Figure 5.3: Test suites created by the SD, SD-choose4, and SD-minimized generation techniques. Fault is the fraction of faults detected. Size is the number of test cases in the suite. All numbers are averaged across 50 suites of each type for each program.

Chapter 6

Why it works

The following two sections explain and justify the insights that led to the specification difference test suite improvement techniques. First, when tests are added at random to a suite, the suite's *specification coverage* increases rapidly, then levels off at a high value. This section defines specification coverage and provides experimental evidence for this claim. Second, Section 6.4 shows that specification coverage is correlated with fault detection, even when test suite size and code coverage are held constant.

6.1 Specification coverage

As discussed in Section 2.1, a specification is a set of assertions about a program, chosen from some grammar. Assume there exists an oracle specification O, containing all true assertions in the grammar. Given a test suite, a specification S can be generated. S contains the assertions that are likely true, based on observations made while running the program over the test suite.

Define $t = |S \cap O|$, the number of true assertions in S. The precision p of S is the fraction of assertions in S that are true, so $p = t/|S|$. The recall r of S is the fraction of assertions in O that are also in S, so $r = t/|O|$. Precision and recall are standard measures from information retrieval [vR79].

We define the specification coverage c of S as the weighted average of precision and recall, giving $c = ap + (1 - a)r$. For simplicity, we choose $a = .5$, giving $c = (p + r)/2$. For example, suppose the oracle contains 10 assertions, and the generated specification contains 12 assertions: 9 true and 3 false. The precision is 9/12 and the recall is 9/10, so the specification coverage is $(9/12 + 9/10)/2 = .825$.

Specification coverage measures the difference between a specification and the oracle. Because it is an average of precision and recall, it accounts for false assertions present in the specification, as well as true assertions missing from the specification. Like other coverage measures, specification coverage is a value between 0 and 1 inclusive, is 0 for an empty test suite, and is 1 for an ideal test suite.

For the purpose of measuring specification coverage, we used Daikon to generate the oracle specifications. The oracle specification is the set of all invariants in Daikon's grammar that are true. This is exactly the specification Daikon would generate, given a good enough test suite. In the extreme, the test suite containing all valid inputs to the program would surely be good enough. We didn't have such test suites, so we used the pool of test cases for each program as an approximation. We believe that the pools are sufficiently large and diverse that adding additional test cases would not change the generated specification.

6.2 Comparing specifications

To calculate specification coverage (Section 6.1), we must first determine the degree of difference between two specifications. This is complicated by invariant justification, filtering of redundant invariants, and invariants of the same type whose constants are filled in differently. This issue is also relevant to the specification difference technique, but less so because we need only determine *if* two specifications differ, not by how much.

As stated in Section 2.2, our specifications are a set of program invariants. Each invariant has several components: its program point, its template, the variables and constants in the template slots, and its probability.

- The program point identifies where in the program the invariant holds. Program points include function entries, function exits, and other locations depending upon the source language of the program.
- The template determines what relationship the invariant can express. Daikon currently includes 35 invariant templates. Examples include `NonZero` (expresses that a variable is never equal to zero) and `LinearBinary` (expresses that one variable is always a linear transformation of another variable).
- Each template has a fixed number of variable and constant slots. An invariant is instantiated by filling the variable and constant slots of its template. For example, the `NonZero` template has one variable

slot and no constant slots, while the `LinearBinary` template has two variable and two constant slots (for $y = ax + b$).

- The probability is Daikon's estimate of how likely the invariant is to happen by chance. It is a number ranging from 0 to 1, with 0 meaning the invariant almost certainly *did not* happen by chance. If the probability is less that some limit (.01 by default), we say the invariant is *justified*. Otherwise, the invariant is *unjustified*. By default, Daikon only prints justified invariants. But for the purpose of comparing specifications, Daikon outputs all invariants regardless of justification.

We determine the difference between two specifications as follows. First, we attempt to pair each invariant in the first specification with the corresponding invariant in the second. Invariants are paired together if they have the same program point, the same template, and the same variables in the template slots. Daikon guarantees that a specification contains at most one invariant for a given program point, template, and set of variables.

If an invariant has no partner in the other specification, and is justified, it counts as a difference of 1. If the invariant is unjustified, it counts as no difference.

Next, we examine each pair of invariants. Since the invariants are paired together, we know they have the same template and variables. The only possible differences are the template constants and the probability. The probability determines whether each invariant is justified or unjustified. There are several possibilities to consider:

- **Same constants, same justification.** The invariants are exactly the same, and either both justified or both unjustified. This counts as no difference.

- **Same constants, different justification.** The invariants are the same, but one is justified and the other is not. This is counted as a difference, but there are two ways to measure the magnitude. It may be counted as a difference of 1, or it can be counted with a magnitude equal to the difference in probabilities. This is further explained later in this section.

- **Different constants, both justified.** The invariants are different, and both are justified. This counts as a difference of 2, since each specification contains an invariant not present in the other specification.

- **Different constants, exactly one justified.** The invariants are different, but only one is justified. This counts as a difference of 1.

- **Different constants, neither justified.** The invariants are different, but neither is justified. This counts as no difference.

This completes the comparison of the specifications, since each invariant has been examined.

We mentioned earlier that if two invariants are the same, but one is justified and the other is not, there are two ways to count the difference. Either count it as a constant difference of 1, or count it with magnitude equal to the difference in probabilities. Accounting for the probabilities is important for some tools that compare specifications. The reason is that many cases may add confidence to an invariant, but no one case may alone justify the invariant. Thus, a tool may need to know when the probability of an invariant changes, even though it remains unjustified.

However, this method is too sensitive for the specification difference technique. Nearly every candidate test case improves the test suite a little bit, so the generated suites are very large. For the SD technique, we ignore specific probabilities, and only consider whether invariants were justified or unjustified.

Another issue is the filtering of redundant invariants. By default, Daikon filters some invariants that are implied by others. For instance, assume Daikon detected the following three invariants: `x == y`, `y == z`, `x == z`. Daikon would not print the last invariant, since it is obviously implied by the first two. This is normally a desirable feature, since the specification is more succinct and easier for people to read. However, this feature is undesirable when comparing specifications. Assume we are comparing the previous specification to a different specification, where `x == z` but the other two invariants are false.

If Daikon filters the redundant invariant, there will be three differences between the specifications, as illustrated in the following table. The second specification is missing two invariants, and has one extra invariant.

Specification 1	Specification 2
`x == y`	
`y == z`	
	`x == z`

If Daikon doesn't filter the redundant invariant, there will be only two differences between the specifications.

Specification 1	Specification 2
`x == y`	
`y == z`	
`x == z`	`x == z`

Clearly, there are only two "real" differences between the two specs. If an invariant in one specification is implied by invariants in the other specification, this shouldn't count as a difference. For each invariant, check if it is logically implied by all the invariants in the other specification. If it is implied, then it doesn't count as a difference. However, checking logical implication of invariants is difficult, so in practice, we handle this issue by disabling Daikon's filtering of redundant invariants. As long as the filtering is disabled for both specifications, the computed difference should be accurate.

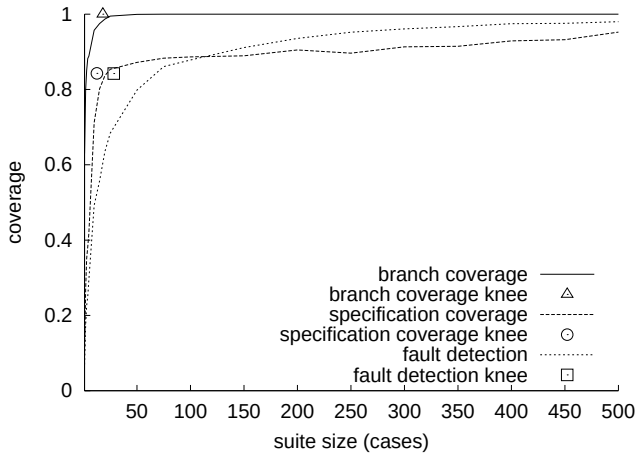

Figure 6.1: Effect of test suite size (measured in cases) on branch coverage, specification coverage, and fault detection of randomly-generated test suites, for the `tot_info` program.

	cases knee		calls knee	
	cases	value	calls	value
stmt. cov.	10	0.96	2971	0.90
spec. cov.	15	0.81	3243	0.80
branch cov.	20	0.94	3409	0.87
fault detection	53	0.74	11796	0.73

Figure 6.2: Table of knee locations, averaged across seven programs. These numbers indicate where plots of statement coverage, specification coverage, fault detection, and branch coverage against time switch from one nearly-linear component to another; they indicate average positions of the knees plotted for one program in Figure 6.1. The first two columns show the x and y coordinates (the number of cases and the height on the graph) of the knee on the cases graph, and last two columns show these values for the calls graph.

6.3 Test suite size

When tests are added at random to a suite, the specification coverage increases rapidly, then levels off at a high value. In other words, there are two distinct types of test suites. Suites inducing relatively poor specifications are measurably improved by almost any augmentation. Suites inducing good specifications are little affected by augmentation, and the specifications are already so close to ideal that they can never be substantially improved.

Figure 6.1 plots average branch coverage, specification coverage, and fault detection against suite size for the `tot_info` program, for 1500 randomly generated test suites of case sizes 1–500. Figure 6.1 does not plot statement coverage because it lies almost exactly under branch coverage.

These data are for the `tot_info` program, but plots for the other programs had similar shapes. In the plots for all eight programs, all measured values rise sharply as suite size increases, then level off to a nearly linear approach towards the asymptote.

Figure 6.1 also plots the knee of each curve. We computed the knee by finding the point that minimizes the summed mean square error of two lines regressed to the sets of points to its left and right. The knee is the intersection of the pair of lines that fit best.

Figure 6.2 gives the average positions of all the knees across all programs. For these programs, statement coverage grows quickest, reaching nearly complete coverage after an average of only 10 tests. Specification coverage levels off only slightly later, but at a lower coverage value. Branch coverage reaches its knee yet later, but at a level closer to that of statement coverage. Fault detection takes the longest to reach its knee, and thereafter continues to rise appreciably even as test suites are made very large.

The coverage curves indicate that good specification coverage is achievable in general, even with random suites of modest size. The knees in the curves support our claim that iterative augmentation results in suites scoring well on the coverage criteria (in particular, specification coverage). The relative locations of the knees indicate the approximate sizes of random suites that one can expect to achieve good specification coverage, compared to those that achieve good code coverage.

Finally, the relatively low height of the specification coverage knee demonstrates that although the inflection point is reached early, there is still room for improvement. Incremental improvements in specification coverage can still be attained, even when statement and branch coverage have reached their maxima.

6.4 Fault detection

Section 6.3 demonstrated that high absolute levels of specification coverage are achievable. This section shows that increasing specification coverage (that is, improving induced specifications) results in greater fault detection. Section 6.4.1 demonstrates the result for arbitrary test suites, and Section 6.4.2 demonstrates that even when a test suite achieves 100% code coverage, increasing specification coverage improves fault detection.

6.4.1 Random suites

We analyzed 1000 randomly generated test suites for each of the eight programs. The suite sizes, in cases, were uniformly distributed between 1 and the number of cases at the fault detection knee (Section 6.3). We did not consider larger suites, because augmenting a large test suite has little effect.

For each test suite, we calculated its size (in cases and calls), statement coverage, branch coverage, specification coverage, and fault detection. Then, we performed six

Independent variable	Dependent variable		
	spec. cov.	stmt. cov.	fault detect.
cases	.285	.037	.250
calls	.068	−.005	.337
spec. cov.	-	.741	.130
stmt. cov.	.593	-	.167

Independent variable	Dependent variable		
	spec. cov.	branch cov.	fault detect.
cases	.169	.162	.229
calls	.075	−.005	.337
spec. cov.	-	.723	.095
branch cov.	.676	-	.234

Figure 6.3: Standardized multiple regression coefficients, averaged across eight programs. Standardized coefficients are the coefficients that would be produced if the data analyzed were in standard score form. "Standard score" form means that the variables have been standardized so that each has a mean of zero and a standard deviation of 1. Thus, standardized coefficients reflect the relative importance of the predictor variables. Each column presents the results from a separate multiple regression.

multiple regressions for each program. This regression indicates how each predictor affects each result, while holding all other factors constant; for example, it avoids conflating the effect of size and coverage, even though larger suites tend to have more coverage.

Each column of Figure 6.3 presents results of one multiple regression. For instance, the upper-left regression uses size and statement coverage as the independent variables, and uses specification coverage as the dependent variable. (We performed two sets of three multiple regressions, rather than one set involving all five variables, because statement coverage and branch coverage are highly correlated; they fail a test of non-collinearity and bias the results. Separating the variables avoids this problem. There is no significant interaction effect among any other predictor variables at the $p = .10$ level.)

The coefficients indicate the direction and relative magnitude of correlation between the independent and dependent variables. For example, statement coverage has a slightly greater standardized effect on fault detection (.167) than does specification coverage (.130). Branch coverage has almost 2.5 times more effect on fault detection (.234) than specification coverage (.095). Statement and branch coverage are too highly correlated to be used in the same regression, so their coefficients were calculated in separate regressions. This means the coefficients for statement and branch coverage cannot be directly compared.

We also computed, but do not show here, raw correlation coefficients. For example, when specification coverage is used to predict fault detection, the coefficient is .340. This means that if specification coverage is increased by 1 percent, and all other predictors are held constant, fault detection increases by .340 percent.

We analyzed the results by examining each dependent variable (column of the tables) separately.

Specification coverage. Code coverage has a large effect on specification coverage. If a statement in the program is never executed, there is no way to see its effect on the specification. Cases and, less significantly, calls have small positive effects on statement coverage.

Statement and branch coverage. Specification coverage has a substantial effect on code coverage. If a specification is covered well, every statement or path that contributes to the specification must be covered. Number of cases has a small effect on statement coverage, and number of calls has almost no effect.

Fault detection. Specification coverage and statement coverage have approximately the same effect on fault detection. Holding all other predictors constant, an increase in statement coverage has about the same effect as an identical increase in specification coverage. Branch coverage has 2.5 times as great an effect as specification coverage, so branch coverage is the better predictor of fault detection. However, cases is as good a predictor as branch coverage, and calls is even more significant: both size measures dwarf statement and specification coverage. Furthermore, whereas cases better predicts specification, statement, and branch coverage, calls is better for fault detection, the measure testers really care about.

There are two important conclusions to draw from this experiment. First, specification coverage is a good predictor of fault detection. Test suites with more specification coverage detect faults better. Second, specification coverage is as good a predictor of fault detection as statement coverage.

6.4.2 Effect of 100% code coverage

A final experiment further demonstrates the value of specification coverage as a test suite quality metric that is independent of code coverage metrics.

For each of the subject programs except `space`, we analyzed 1000 suites with statement coverage, 1000 suites with branch coverage, and 1000 suites with def-use coverage. For `space`, we only analyzed 1000 suites with branch coverage. (We obtained the suites from Rothermel and Harrold [RH98]; there were no statement or def-use covering suites for `space`, nor were we able to generate them.) Section 4.1 describes these test suites, and Figure 4.4 presents their average sizes in cases. The statement and branch coverage suites have about the same number of cases, while the def-use coverage suites are three times as large.

We calculated the size, specification coverage, and fault detection rate of each test suite. For each type of coverage and each program, we performed a multiple regression, with size and specification coverage as the independent variables and fault detection as the dependent variable.

Coverage type	Spec. cov. coefficient	Mean spec. cov.	Mean fault detect	# stat. sig.	# not sig.
statement	.483	.877	.396	5	2
branch	.308	.866	.461	6	2
def-use	.507	.950	.763	2	5

Figure 6.4: Multiple regression coefficient for specification coverage, when regressed against fault detection. The coefficient for size was not statistically significant for any of the programs, and has been omitted from the table. The coefficient for specification coverage was only statistically significant for some of the programs. The "# stat. sig." column contains this number, and the "# not sig." column contains the number of programs for which the coefficient was not statistically significant. Each value was averaged across all programs for which the specification coverage coefficient was statistically significant.

We performed 21 multiple regressions in total (7 programs $\times$ 3 coverage criteria). Figure 6.4 summarizes the results.

The coefficient describes the relationship between specification coverage and fault detection. For example, the coefficient of .48 for statement coverage suites suggests that if the specification coverage of a suite were increased by 1 percent, and all other factors held constant, the fault detection rate would increase by approximately .48 percent.

The mean specification coverage and fault detection indicate how much improvement is possible, since their maximum values are 1.

These results show that, for test suites with branch or statement coverage, increasing specification coverage does increase fault detection. However, for suites with def-use coverage, fault detection is often independent of specification coverage (only 2 programs had statistically significant coefficients). This might be because specification coverage is already near perfect for those test suites.

Further, these results show that specification coverage is complementary to code coverage for detecting faults. Even when statement or branch coverage is 100%, increasing specification coverage can increase the fault detection of a test suite without increasing test suite size. Stated another way, specification coverage indicates which of otherwise indistinguishable (with respect to code coverage) test suites is best.

Chapter 7

Related work

This work builds on research in specification-based testing. For the most part, that research has focused on systematic generation, not evaluation, of test suites, and has required users to provide a specification a priori.

7.1 Specifications for test suite generation

Goodenough and Gerhart [GG75b, GG75a] suggest partitioning the input domain into equivalence classes and selecting test data from each class. Specification-based testing was formalized by Richardson et al. [ROT89], who extended implementation-based test generation techniques to formal specification languages. They derive test cases (each of which is a precondition–postcondition pair) from specifications. The test cases can be used as test adequacy metrics. Even this early work emphasizes that specification-based techniques should complement rather than supplement structural techniques, for each is more effective in certain circumstances.

The category-partition method [OB88] calls for writing a series of formal test specifications, then using a test generator tool to produce tests. The formal test specifications consist of direct inputs or environmental properties, plus a list of categories or partitions for each input, derived by hand from a high-level specification. Balcer et al. [BHO89] automate the category-partition method for writing test scripts from which tests can be generated, obeying certain constraints. These specifications describe tests, not the code, and are really declarative programming languages rather than specifications. While their syntax may be the same, they do not characterize the program, but the tests, and so serve a different purpose than program specifications.

Donat [Don97] distinguishes specifications from test classes and test frames and gives a procedure for converting the former into each of the latter (a goal proposed in earlier work [TDJ96]), but leaves converting the test frames into test cases to a human or another tool. Dick and Faivre [DF93], building on the work of Bernot et al. [BGM91], use VDM to generate test cases from preconditions, postconditions, and invariants. Meudec [Meu98] also uses a variety of VDM called VDM-SL to generate test sets from a high-level specification of intended behavior. Offutt et al. generate tests from constraints that describe path conditions and erroneous state [Off91] and from SOFL specifications [OL99].

All of this work assumes an a priori specification in some form, and most generate both test cases and test suites composed of those test cases. By contrast, the specification difference technique assumes the existence of a test case generator (any of the above could serve), then generates both a test suite and a specification.

7.2 Specifications for test suite evaluation

Chang and Richardson's structural specification-based testing (SST) technique [CR99] uses formal specifications provided by a test engineer for test selection and test coverage measurement. Their ADLscope tool converts specifications written in ADL [HS94] into a series of checks in the code called coverage condition functions [CRS96]. Once the specification (which is about as large as the original program) is converted into code, statement coverage techniques can be applied directly: run the test suite and count how many of the checks are covered. An uncovered test indicates an aspect of the specification that was inadequately exercised during testing. The technique is validated by discovery of (exhaustively, automatically generated) program mutants.

This work is similar to ours in that both assume a test case generation strategy, then evaluate test suites or test cases for inclusion in a test suite. However, SST requires the existence of an a priori specification, whereas the specification difference technique does not, but provides a specification.

7.3 Related coverage criteria

Several researchers have proposed notions of coverage that are similar to our definition of specification coverage (Section 6.1). In each case, a test suite is evaluated with respect to a goal specification. The related work

extends structural coverage to specifications, computing how much of the specification is covered by execution of the test suite. By contrast, our definition compares a generated specification to the goal specification. Our specification coverage is generally harder to satisfy, because the specification generator's statistical tests usually require multiple executions before outputting a specification clause, whereas structural coverage requires only one satisfying execution.

Burton [Bur99] uses the term "specification coverage" to refer to coverage of statements in a specification by an execution; this concept was introduced, but not named, by Chang and Richardson [CR99]. Burton further suggests applying boolean operand effectiveness (modified condition/decision coverage or MC/DC) to reified specifications; this coverage criterion requires that each boolean subterm of a branch condition take on each possible value. Other extensions of structural coverage criteria to specification checks are possible [Don97] but have not been evaluated.

Hoffman et al. [HSW99, HS00] present techniques for generating test suites that include tests with (combinations of) extremal values. These suites are said to have boundary value coverage, a variety of data coverage. The Roast tool constructs such suites and supports dependent domains, which can reduce the size of test suites compared to full cross-product domains. Ernst [Ern00] uses the term value coverage to refer to covering all of a variable's values (including boundary values); the current research builds on that work.

Hamlet's probable correctness theory [Ham87] calls for uniformly sampling the possible values of all variables. Random testing and operational testing are competitive with or superior to partition testing, debug testing, and other directed testing strategies, at least in terms of delivered reliability [DN84, HT90, FHLS98, FHLS99]. This work makes several reasonable assumptions such as that testers have good but not perfect intuition and that more than a very small number of tests may be performed. Specification coverage is likely to assist in both operational and random testing, permitting improved test coverage, and better understanding of the test cases, in both situations.

Chang and Richardson's operator coverage [CR99] is not a measure of test suite quality, but concerns the creation of mutant (faulty) versions of programs. Operator coverage is achieved if every operator in the program is changed in some mutant version. The mutants can be used to assess test suite comprehensiveness, in terms of fault detection over the mutants.

Amman and Black [AB01] measure test suite coverage in terms of number of mutant specifications (in the form of CTL formulae) killed. A mutant version of a specification contains a specific small syntactic error, and a test suite is said to kill the mutant if the test suite gives a different result over the faulty version than it does over the correct version. Amman and Black use model checking to check the test cases against the CTL specifications. If every mutant is killed, then every component of the specification is covered, since every component of the specification was mutated.

Chapter 8

Future work

We feel there is great promise for using generated specifications for testing activities. This research focuses on one application of specifications: the specification difference (SD) technique for generating, augmenting, and minimizing test suites.

The future work falls into two categories. First, there remain many questions about the SD technique. Second, there are many other potential applications of generated specifications to testing.

8.1 Specification difference technique

8.1.1 Additional sample programs

Further experimental validation is required to extend the results of Chapters 4–6, which might not generalize to other programs or test suites. The preliminary subject programs are quite small, and larger programs might have different characteristics. (We have some confidence this is not the case, since the `space` program was 10 times larger than any other subject, yet had similar characteristics.) The main reason to choose these subject programs is their suite of tests and faulty versions. We did not have access to other programs with human-generated tests and faulty versions, and we suspect that our subject programs differ from large programs less than machine-generated tests and faults differ from real ones. The experiments of Chapters 4–6 should be performed on additional large programs, to increase our confidence in and the generality of our results.

Similarly, all of our subject programs are written in the C programming language. However, Daikon can generate specifications for programs in many languages, including C and Java. We believe our results will apply to Java as well as C, and we could verify this by running experiments on Java programs.

8.1.2 Properties of specifications

A key property of any specification is its level of detail. A specification that is too detailed may be useless, because the important concepts are buried in trivial facts. Similarly, a specification with too little detail may not provide the information you seek. There is no right or wrong amount of detail. Different tasks require different levels of detail. What level of detail is best for specifications used in the SD technique? Can the technique be improved by using more or less detailed specifications?

A related question involves the specifications produced at intermediate steps in the SD technique. Adding a test case to a test suite can change the suite's specification in two ways. First, the new test can falsify a property that was true in the rest of the suite but is not true in general. Second, the new test can reinforce a property that is true over the rest of the suite, increasing its statistical justification so that it is included in the output specification. How does the generated specification change throughout the SD technique? Are more properties added or removed? We plan to perform more extensive experiments to characterize this phenomenon, leading to a theoretical model and possibly to improved statistical tests or a better test suite generation technique.

8.1.3 Improve performance

Chapter 5 described enhancements that improve the fault detection of suites generated by the SD technique. Future work should also consider improving the technique's performance. The technique requires frequent generation of specifications, and dynamic specification generation is computationally expensive, at least with current technology. More efficient specification generation techniques will surely be discovered, but the easiest way to improve the performance of the SD technique is to reduce the number of times specifications must be generated. A possible tactic is simultaneously adding multiple test cases, instead of adding one case at a time. Experiments should be performed to investigate the relationship between runtime and quality of output.

8.1.4 Generate specification-covering suites

In Section 6.1, we showed that test suites generated by the SD technique will have a high level of specification coverage. However, it should be possible to create test suites

with 100% specification coverage. We have begun preliminary investigations into creating specification-covering suites using a technique similar to specification difference. Unfortunately, the generated suites are very large, averaging 478 cases (compared to 32 cases for the original SD suites). We hypothesize that smaller test suites can achieve 100% specification coverage. We haven't examined the generated suites in detail, but we speculate that a few difficult-to-justify invariants are inflating the test suites. We plan to tweak the statistical algorithms to make certain invariants easier to justify.

8.1.5 Improve the evaluation

In Section 4.1, we compare the SD technique to code coverage techniques in terms of size and fault detection of generated test suites. This comparison is slightly unfair to the SD technique — the suites generated by the code coverage techniques have 100% code coverage, but the suites generated by the SD technique have less than 100% specification coverage.

Code coverage is defined as the number of executed code features (statements, branches, or def-use pairs) divided by the number of reachable code features. However, a tester in the field doesn't know which code features of a program are reachable. The tester would probably take an approach similar to the SD technique: add cases to a suite until coverage stops increasing. How different is such a suite from a suite with 100% code coverage?

To be fair, we should evaluate the SD technique against these code coverage techniques. Alternatively, we could evaluate suites with 100% specification coverage (Section 8.1.4) against the suites with 100% code coverage.

8.1.6 Integrate with test case generator

A limitation of the SD generation and augmentation techniques is the requirement of a test case generator. Our experiments used a pool of test cases as a generator, but test cases may be generated at random or from a grammar, created by a human, or produced in any other fashion. Perhaps the best approach is to build upon work in automatic generation of test cases and code-covering test suites. One specific approach we have begun investigating at MIT is a system called Korat that generates input instances from a boolean method that reports whether an instance is legal [BKM02]. Given a `check()` method that returns true or false, Korat examines the space of all possible instances; when an instance is rejected, Korat uses the structure of the `check()` method to prune the search space. Initial results suggest that the resulting exhaustive set of small instances is often, but not always, appropriate as a test suite. We expect that our work can be integrated with other work on generating test cases as well.

8.2 Other applications

We believe that generated specifications can benefit many other testing tasks. This thesis presents a technique for generating, augmenting, and minimizing test suites. The generation and augmentation techniques require a test case generator, and as such are test case *selection* techniques, not test case generation techniques.

However, we believe it may be possible to suggest new test cases by examining properties of the specifications. For instance, a property may be true for all executions seen thus far, but it may not be statistically justified. A tool could suggest test cases that would either falsify or add confidence to this property. Or, a tool could suggest test cases that probe the boundary values of the current generated specification.

Generated specifications could be used as a form of regression testing, by comparing the specifications generated from two versions of a program. If the specifications differ in an unintended way, there is a good chance the programs also differ in an unintended way, even if their outputs happen to be the same. Similarly, if the specifications are the same, there is a good chance the programs behave the same. A generated specification could never replace a regression testing suite, but it could be a useful complement. A specification may contain properties that were overlooked in the test suite. Further investigation should be performed to determine the utility of specifications as a regression testing aid.

We demonstrated that both code coverage (of various sorts) and specification coverage are correlated to fault detection (Section 6.4), and that improving specification coverage tends to detect different faults than improving code coverage does (Section 4.4.1). However, this does not indicate how much relative effort a software tester should invest in improving the two types of coverage. We plan to assess how difficult it is to the programmer to increase specification coverage, relative to the work it takes to achieve a similar gain in statement coverage. Also, the last few percent of code coverage are the hardest to achieve; is specification coverage similar?

A final major goal of this research is to evaluate the use of generated specifications in performing programming tasks. We will investigate this question by observing how people use our tools. This will both answer questions about them and also point out opportunities to improve them. We anticipate that our investigations will focus on case studies rather than controlled experiments. Because our goal is to support the proposition that generated specifications can be valuable, examining a broader range of tasks and providing evidence from case studies is more appropriate at this stage of investigation.

Chapter 9

Conclusion

We have proposed and evaluated the specification difference technique for generating, augmenting, and minimizing test suites. The technique selects test cases by comparing specifications dynamically generated from test suites. A test case is considered important to a test suite if its addition or removal causes the specification to change. The technique is automatic, but assumes the existence of a test case generator that provides candidate test cases.

The specification difference (SD) generation technique performs about as well as the branch coverage technique, but is better at detecting certain types of faults. Combining the SD augmentation technique with branch coverage yields better results than either technique alone. The SD minimization technique performs better than branch coverage minimization, but the suites are larger in size. The technique also generates an accurate specification for the program under test, which has many benefits in itself.

We have described a new form of specification coverage that uses statistical tests to determine when a specification clause has been covered. We justified the success of the SD technique using additional experiments relating specification coverage to size and fault detection. Specification coverage can be used on its own to evaluate any existing test suite.

Finally, the work presented in this thesis is just one application of generated specifications. We believe that generated specifications hold similar promise for other areas of testing research.

Acknowledgments

I am grateful to my advisor Michael Ernst, for his technical and personal guidance over the past year. I also thank the members of the Program Analysis Group — particularly Ben Morse, Jeremy Nimmer, and Alan Donovan. Special thanks goes to Ben for creating the C front end for Daikon, which made these experiments possible. Thanks to Jeremy and Alan for commenting on various aspects of this research, and answering questions whenever they came up. Thanks to Steve Wolfman, Orna Raz, Gregg Rothermel, Steve Garland, and anonymous referees for suggestions on previous papers describing this research. Finally, thanks to Vibha Sazawal for providing statistical consulting.

This research was supported in part by NTT, Raytheon, an NSF ITR grant, and a gift from Edison Design Group.

Bibliography

[AB01] Paul E. Amman and Paul E. Black. A specification-based coverage metric to evaluate test sets. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):275–299, 2001.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, Boston, MA, third edition, 2000.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6, November 1991.

[BHO89] Marc J. Balcer, William M. Hasling, and Thomas J. Ostrand. Automatic generation of test scripts from formal test specifications. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 210–218, 1989.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, July 22–24, 2002.

[Bur99] Simon Burton. Towards automated unit testing of statechart implementations. Technical report, Department of Computer Science, University of York, UK, 1999.

[CR99] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 285–302, September 6–10, 1999.

[CRS96] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 62–70, 1996.

[DF93] J. Dick and A. Faivre. Automating the generating and sequencing of test cases from model-based specifications. In *FME '93: Industrial Strength Formal Methods, 5th International Symposium of Formal Methods Europe*, pages 268–284, 1993.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[Don97] Michael R. Donat. Automating formal specification-based testing. In *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, pages 833–847. Springer-Verlag, April 1997.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE'00, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE'99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[EGKN99] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA, November 16, 1999. Revised March 17, 2000.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, Uni-

versity of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[FHLS98] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, August 1998. Special Section: International Conference on Software Engineering (ICSE '97).

[FHLS99] Phyllis Frankl, Dick Hamlet, Bev Littlewood, and Lorenzo Strigini. Correction to: Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 25(2):286, March/April 1999.

[GG75a] John B. Goodenough and Susan L. Gerhart. Correction to "Toward a theory of test data selection". *IEEE Transactions on Software Engineering*, 1(4):425, December 1975.

[GG75b] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1 edition, 1991.

[GL00] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[Ham87] Richard G. Hamlet. Probable correctness theory. *Information Processing Letters*, 25(1):17–25, April 20, 1987.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 16–21, 1994.

[HS94] Roger Hayes and Sriram Sankar. Specifying and testing software components using ADL. Technical Report TR-94-23, Sun Microsystems Research, Palo Alto, CA, USA, April 1994.

[HS00] Daniel Hoffman and Paul Strooper. Tools and techniques for Java API testing. In *Proceedings of the 2000 Australian Software Engineering Conference*, pages 235–245, 2000.

[HSW99] Daniel Hoffman, Paul Strooper, and Lee White. Boundary values and automated component testing. *Software Testing, Verification, and Reliability*, 9(1):3–26, March 1999.

[HT90] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM'01, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[Lam88] David Alex Lamb. *Software Engineering: Planning for Change*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[LG01] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA, 2001.

[Meu98] Christophe Meudec. *Automatic Generation of Software Test Cases From Formal Specifications*. PhD thesis, Queen's University of Belfast, 1998.

[NE01a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation and checking of program specifications. Technical Report 823, MIT Laboratory for Computer Science, Cambridge, MA, August 10, 2001. Revised February 1, 2002.

[NE01b] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[Off91] A. Jefferson Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[OL99] A. Jefferson Offutt and Shaoying Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62, December 1999.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[Pre92] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, third edition, 1992.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[ROT89] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to specification-based testing. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 86–96, December 1989.

[Sem94] Semiconductor Industry Association. The national technology roadmap for semiconductors. San Jose, CA, 1994.

[Som96] Ian Sommerville. *Software Engineering*. Addison-Wesley, Wokingham, England, fifth edition, 1996.

[TDJ96] Kalman C. Toth, Michael R. Donat, and Jeffrey J. Joyce. Generating test cases from formal specifications. In *6th Annual Symposium of the International Council on Systems Engineering*, Boston, July 1996.

[VF98] Filippos I. Vokolos and Phyllis G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings; International Conference on Software Maintenance*, pages 44–53. IEEE Computer Society Press, 1998.

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(3), February 2002.