Improving Test Suites via Operational Abstraction

Michael Harder Jeff Mellen Michael D. Ernst

MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{mharder,jeffm,mernst}@lcs.mit.edu

Abstract

This paper presents the operational difference *technique for generating, augmenting, and minimizing test suites. The technique is analogous to structural code coverage techniques, but it operates in the semantic domain of program properties rather than the syntactic domain of program text.*

The operational difference technique automatically selects test cases; it assumes only the existence of a source of test cases. The technique dynamically generates operational abstractions (which describe observed behavior and are syntactically identical to formal specifications) from test suite executions. Test suites can be generated by adding cases until the operational abstraction stops changing. The resulting test suites are as small, and detect as many faults, as suites with 100% branch coverage, and are better at detecting certain common faults.

This paper also presents the area *and* stacking *techniques for comparing test suite generation strategies; these techniques avoid bias due to test suite size.*

1 Introduction

Program specifications play a valuable role in dynamic analyses such as software testing [GG75b, ROT89, CRS96, OL99]. Most previous research on automated specification-based testing has required software testers to provide a formal specification. Furthermore, the research has generally focused on systematic generation of test suites rather than evaluation and comparison of existing test suites. The applicability of previous research is limited by the fact that very few programs are formally specified — most lack even `assert` statements. Additionally, software engineers may desire to improve existing test suites or to use other tools for selecting test cases.

The *operational difference* (OD) technique is a new specification-based test case selection technique that enables augmentation, minimization, and generation of test suites in cooperation with any existing technique for test case generation. It does not require a specification to be provided *a priori*, nor does it require an oracle that indicates whether a particular test case passes or fails. It automatically provides users with an *operational abstraction*. An operational abstraction is syntactically identical to a formal specification, but describes actual behavior, which may or may not be desired behavior. For future testing, the operational abstraction can provide the benefits of an *a priori* specification, such as assisting regression testing by revealing removed functionality and checking test results. The operational abstractions are also useful in their own right.

This research builds on previous testing research, such as structural coverage criteria, but applies it to the semantic domain of program properties rather than the syntactic domain of program text. The key idea behind the operational difference technique is comparison of operational abstractions, which are dynamically generated from program executions. A test case that improves the operational abstraction can be added to a test suite; a test case that does not can be removed, if appropriate. We have implemented the technique and experimentally demonstrated its efficacy in creating test suites with good fault detection.

The remainder of the paper is organized as follows. Section 2 describes how to automatically induce an operational abstraction from a program and a test suite. Section 3 presents the operational difference technique for generating, augmenting, and minimizing test suites. Section 4 details our experimental methodology, and Section 5 reports experiments demonstrating that the operational difference technique compares favorably with other methods. Section 6 provides intuition (and experimental data) for why the technique works. Section 7 summarizes related research, then Section 8 concludes.

2 Generating operational abstractions

An *operational abstraction* is a formal mathematical description of program behavior: it is a collection of logical statements that abstract the program's runtime operation. An operational abstraction is identical in form to a formal specification. However, a formal specification is intended to be written before the code and to express desired behavior.

By contrast, an operational abstraction expresses observed behavior and is induced dynamically from program executions.

Our experiments use the Daikon dynamic invariant detector to generate operational abstractions, but the ideas generalize beyond any particular implementation. Briefly, Daikon 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. The output is improved by suppressing invariants that are not statistically justified and by other techniques [ECGN00].[1] Dynamic invariant detection is not affected by the internal structure of the program component being analyzed. In Zhu's terminology [ZHM97], our technique is interface-based, which is a variety of black-box testing.

Generation of operational abstractions from a test suite is unsound: the properties are likely, but not guaranteed, to hold in general. 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 cases. 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, but this paper does not address that issue.

Operational abstractions often match human-written formal specifications [Ern00, ECGN01] or can be proved correct [NE02a, NE02b, NWEG$^+$03]. However, the operational difference technique does not depend on these properties: it generates test suites that detect faults even if the operational abstraction differs from the formal specification that a human might have written, and even if the program is incorrect.

3 Operational difference technique

This section describes the operational difference (OD) technique for generating, augmenting, and minimizing test suites. The OD technique compares the operational abstractions induced by different test suites in order to decide which test suite is superior.

The operational difference technique is automatic, but for test suite generation and augmentation, it assumes the existence of a source of test cases. In other words, the technique selects, but does not generate, test cases — much like other techniques such as those based on code coverage. Test cases may be created by a human, generated at random or from a grammar, generated from a specification, extracted from observed usage, or produced in any other fashion.

```
procedure OD-GENERATE (program $P$, int $n$)
  testsuite $S \leftarrow \{\}$
  int $i \leftarrow 0$
  while $i < n$ do
    testcase $c \leftarrow$ NEWCASE()
    if OPABSTR($P$, $S$) $\neq$ OPABSTR($P$, $S \cup \{c\}$) then
      $S \leftarrow S \cup \{c\}$
      $i \leftarrow 0$
    else
      $i \leftarrow i + 1$
  return $S$
```

Figure 1. Pseudocode implementation of the basic operational difference (OD) 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. OPABSTR is a procedure that generates an operational abstraction from a program and a test suite; our experiments use the Daikon invariant detector (Section 2).

The basic OD generation technique (Figure 1) starts with an empty test suite and an empty operational abstraction. It repeatedly adds candidate test cases; if they change the operational abstraction, they are retained, under the (possibly incorrect) assumption that an operational abstraction generated from a larger test suite is better. When n candidate cases have been consecutively considered and rejected, the process is complete. The value chosen for n is a tradeoff between the running time of the generation process and the quality of the generated suite; the quality of the test case generation procedure may also influence the best choice for n. We further improve the basic OD generation technique by evaluating several candidate test cases at a time and selecting the case that changes the operational abstraction most, and by minimizing the resulting test suite after generation. The test suite augmentation technique is identical but starts with a non-empty test suite.

The OD test suite minimization technique considers each test case in turn and removes each one whose removal does not change the operational abstraction. Its intuition is that test cases that affect the operational abstraction are more different from one another than test cases that do not affect the operational abstraction. Minimization is similar to generation in that both select a subset of test cases. Minimization differs in that it starts from a suite and removes cases, it considers every case in the suite, and the resulting suite is guaranteed to have the same qualities (such as coverage or operational abstraction) as the original suite. Thus, minimization and generation typically produce quite different suites. Minimization can be useful after generation, because later test cases may subsume earlier ones.

[1] The statistical tests use a user-settable confidence parameter. The results in this paper use the default value, .01. We repeated the experiments with values of .0001 and .99, and the differences were negligible.

Candidate	Operational abstraction for `abs`		Final
test case	Precond.	Postcondition	suite
5	`x == 5`	`x == return`	√
1	`x >= 1`	`x == return`	√
4	*same*	*same*	

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

-6	*empty*	`(x >= 1) => (x == return)`
		`(x <= -1) => (x == -return)`
		`return >= 1`
			√

-3	*same*	*same*	

0	*same*	`(x >= 0) => (x == return)`
		`(x <= 0) => (x == -return)`
		`return >= 0`
			√

7	*same*	*same*	
-8	*same*	*same*	
3	*same*	*same*	

Figure 2. Example of generating a test suite via the OD-base technique. The candidate test cases are considered in order from top to bottom; a test case is added to the suite if it changes the operational abstraction. The process terminates when $n = 3$ test cases have been consecutively considered and rejected. The final test suite is $\{5, 1, -1, -6, 0\}$. The final operational abstraction has no preconditions and three postconditions. All postconditions contain `x = x'` (that is, `x` is not modified), so it is omitted for brevity.

As with many minimization techniques, the OD technique does not guarantee that the minimized suite is the smallest set of cases that can generate the original operational abstraction. Furthermore, there is no guarantee that the removed test case cannot detect a fault. A software engineer might use minimization on test suites of questionable value, such as ones that were randomly generated. Minimization, like test case prioritization, can also be valuable in reducing a test suite. For instance, a large test suite might run every night or every weekend, but developers might run a subset of the suite more frequently.

Statistical tests in the current implementation make the operational abstractions a multiple-entity criterion [JH01]: a single test case may not guarantee that a particular invariant is reported or not reported. This is not a necessary condition of the technique, and we plan to experiment with ways to combine operational abstractions for individual test cases, which will reduce the cost of suite generation.

Figure 2 illustrates how the OD technique with $n = 3$ generates a test suite for the absolute value procedure. The example uses an operational abstraction generator that reports the following properties at procedure entries and exits:

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

4 Experimental methodology

This section describes the subject programs we used to evaluate the operational difference technique, the measurements we performed, and how we controlled for size when comparing test suite generation strategies.

4.1 Subject programs

Our experiments analyze eight C programs. Each program comes with faulty versions and a pool of test cases. (We discovered additional errors in four of the programs during the course of this research.) Figure 3 lists the subjects.

The first seven programs in Figure 3 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 outputs of the faulty and correct versions differ.)

The eighth program, `space`, interprets Array Definition Language inputs. 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 in the control flow graph 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 generated by randomly 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 that there are no statement or def-use covering suites for `space`.

Threats to validity. Our subject programs are of moderate size; larger programs might have different characteristics. We chose them because they are well-understood from previous research and because we did not have access to other programs with human-generated tests and faulty versions. We suspect that these programs differ from large programs

	Program size			Faulty	Test pool			Description
Program	Procedures	LOC	NCNB	versions	cases	calls	op. abs. 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 3. 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 dynamic non-library procedure calls. Operational abstraction size is the number of invariants generated by the Daikon invariant detector when run over the program and the full test pool.

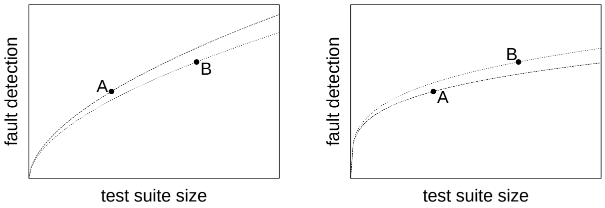

Figure 4. Pitfalls of comparing test suites of different sizes. The labeled points indicate the natural size and fault detection of suites produced by test suite generation techniques A and B. In the two graphs, the natural test suites are identical. The curves show the size and fault detection of "stacked" suites generated by randomly combining or subsetting the natural suites. In the left-hand graph, technique A is superior. In the right-hand graph, technique B is superior.

less than machine-generated tests and faults differ from real ones.

The pool of test cases was generated from test specification scripts, then augmented with code coverage in mind. Thus, the pool may be relatively better for code coverage techniques than for the operational difference technique. The faults were selected based on how many cases detected the fault. This has an undetermined effect, particularly since previous researchers had eliminated the hardest faults, detected by only a few test cases. The small number of faults for certain programs quantized the fault detection ratio to a small number of values, making it harder to correlate with other measures. The combination of few statements — sometimes only 100 — and high coverage similarly quantized statement coverage.

4.2 Measurement details

This section describes some of the measurements we performed. Others, such as code size, are standard.

Test suite size. We measured test suite size in terms of test cases and procedure 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 input to the program under test. This is the most readily apparent static measure of size. However, a single case might execute only a few machine instructions or might run for hours.

The number of dynamic non-library procedure calls approximates the runtime required to execute the test suite, which is likely to be more relevant to testers and which has a strong effect on fault detection. We always measured both quantities, but we sometimes present results for only one of them if the other has a similar effect.

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 number of statements or branches covered by the full set of test cases.

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 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 the faulty versions.

4.3 Comparing test suites via stacking and area

It is misleading to directly compare test suite generation techniques that create suites of different sizes, because test suite size strongly affects fault detection (see Section 6.3).

Comparing test suites using the fault detection to size ratio (also known as "efficiency") is unsatisfactory because the fault-detection-versus-size curve is not a straight line (see Figure 4): doubling size does not double fault detection, and any such curve has points of arbitrarily low efficiency. More seriously, efficiency does not address the question of what technique to use to detect the greatest number of faults for a given test budget (in terms of suite execution time). We propose the *area* technique (described below) to compare test suite generation strategies, rather than simply comparing test suites.

We first observe that even if a test suite generation strategy produces suites of a particular size (call it the natural size), a tester can make suites of arbitrary runtime to fit the testing runtime budget by a process we call *stacking*. If the generated suite is too large, the tester can use a subset of it — if nothing else, a random subset. If the generated suite is too small, the tester can combine suites generated by the technique until they reach the goal size, choosing a subset of the last generated suite. (We assume the test suite generator is not deterministic, so the suites being combined are not identical. For instance, the test suite generator may select test cases, and candidate test cases are presented to it in different orders to create different suites.)

The stacked suites may not be as good as suites directly generated to be of equivalent size. For example, it may be better to generate a suite that covers each statement twice than to stack single-statement-coverage suites to the size of the twice-coverage suites. However, directly generating to an arbitrary size may not be available to testers, and testers may not know which technique to use to create a suite that exactly fits their testing budget. Testers can use stacking with any available test suite generator.

Stacking permits comparison of different test suite generation strategies at arbitrary test suite sizes. Comparing at any particular size might disadvantage one strategy or the other, and different projects have different testing budgets, so it is necessary to compare the techniques at multiple sizes. We sample from fault-detection-versus-size curves induced by stacking, then compare the areas under the curves. This gives an idea of how the techniques are likely to compare at an arbitrary size. More specifically, we create the curves by stacking each test generation strategy to the natural sizes of all the others. In practice, the curves do not cross, so the sense (but not the magnitude) of the comparison could be determined from any single size. The fact that the curves do not cross suggests that stacking tends to preserve the qualities of test suites.

5 Evaluation

This section experimentally evaluates the operational difference (OD) technique for generating, augmenting, and minimizing test suites.

5.1 Test suite generation

We compared test suites generated by the operational difference, statement coverage, branch coverage, def-use coverage, and random selection techniques.

	OD		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
Partial average	.460	13.2	.380	9.9	.425	10.9	.670	29.8
`space`	.795	62.5	–	–	.909	155.2	–	–
Average	**.502**	**19.3**	–	–	**.486**	**28.9**	–	–

Figure 5. Test suites created via automatic generation techniques. "Fault" is the fraction of faults detected. "Size" is the number of test cases in the suite; similar results hold for size measured in terms of number of procedure calls (runtime). All numbers are averaged across 50 suites of each type for each program. Statement and def-use coverage suites were not available for `space`.

Technique	Ratio
Def-use coverage	1.73
Branch coverage	1.66
Operational difference	1.64
Statement coverage	1.53
Random	1.00

Figure 6. Effectiveness of test suite generation techniques, measured by the area technique of Section 4.3, compared to random selection.

We arbitrarily chose $n = 50$ for the OD technique (Figure 1), meaning the process is terminated when 50 consecutive test cases have been considered without changing the operational abstraction. A different value for n might change generation time or suite quality. The structural coverage suites were generated to achieve perfect coverage; we do not know how many candidate test cases were rejected in a row while generating them.

For each subject program, we generated 50 test suites using the OD technique and measured their average size and fault detection. Figure 5 compares them to other automatically generated suites. These include suites achieving complete statement, branch, and def-use coverage. The "average" line shows that the OD test suites are, on average, 2/3 the size of the branch-covering test suites, but achieve slightly better fault detection (.502 versus .486).

In order to better quantify the differences among the test generation techniques, we employed the area and stacking methodology (Section 4.3) to compare them. We generated operational difference, statement coverage, branch coverage, and def-use coverage suites. Then, we stacked (different) OD, statement, branch, def-use, and random suites to each of their natural sizes, for a total of 20 suites at 4 sizes, describing 5 fault-detection-versus-size curves. We repeated the process 50 times for each program, for a total of approximately 2000 fault detection curves (`space` had no statement or def-use curves). We summed the area under all of the curves for a given technique, then computed the ratios of the sums. Figure 6 presents the results: def-use performs best, followed by branch and OD (which are nearly indistinguishable), statement, and random.

Constructing structural coverage suites is difficult, tedious, and often impractical. Test cases that covered each structural element at least 30 times were generated by hand (see Section 4.1). By contrast, no special test cases were provided for the OD technique.

5.1.1 Detecting specific faults

The operational difference technique provides about the same fault detection as branch coverage according to the area technique; this is more accurate than simply noting that on average the OD suites are smaller, and have better fault detection, than the branch coverage suites. Additionally, the OD technique is superior for detecting certain types of faults.

We compared the individual fault detection rates for the OD technique and the 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.

There are a total of 163 faults in the 8 programs we examined. The OD 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 wished to determine whether there is a qualitative difference between the faults detected best by the OD technique and the faults detected best by the stacked branch coverage technique. We examined each faulty version by hand to determine whether it 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.

	OD better	Same	Branch better	Total
CFG change	9	11	9	29
Non-CFG change	56	54	24	134
Total	65	65	33	163

The OD technique is better at detecting value (non-CFG) changes. This makes intuitive sense, because our operational abstractions make assertions about the values of variables. The techniques are equally good at detecting CFG changes. (We expected branch coverage to dominate in this case, because it is measured in terms of the CFG.) Finally, in our target programs, the non-CFG change faults outnumber the CFG change faults by a factor of 4.6. The faults were real faults or were created by people who considered them realistic and representative, so the distribution of faults in practice may be similar.

5.2 Test suite augmentation

The operational difference augmentation technique is identical to the generation technique, except the process is started with an existing test suite rather than an empty test suite. We evaluated this technique by starting with branch coverage suites; the following table gives the results of evaluating them using the area technique. (The numbers differ slightly from previous results because we did not compare at the def-use natural size, and because this experiment used the basic rather than the improved OD technique. The raw numbers are available in a technical report [Har02].)

Technique	Ratio
Branch coverage	1.70
Operational difference	1.72
Branch + operational difference	2.16
Random	1.00

As suggested in Section 5.1.1, code coverage and operational coverage techniques are complementary: combining them is more effective than using either in isolation.

5.3 Test suite minimization

For each program, we generated 50 random test suites with 100 cases each. We minimized these by the OD technique and by maintaining branch coverage; this table gives the results, as measured by the area technique.

Technique	Ratio
Branch coverage	1.50
Operational difference	1.21
Random	1.00

Figure 7 shows some additional data: the OD technique results in better fault detection, but also substantially larger test suites, than minimizing while maintaining branch coverage.

6 Why it works

This section explains and justifies the insights that led to the operational difference test suite improvement techniques. Section 6.1 defines operational coverage, which measures the quality of an arbitrary operational abstraction

	Orig		OD-min		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 7. 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.

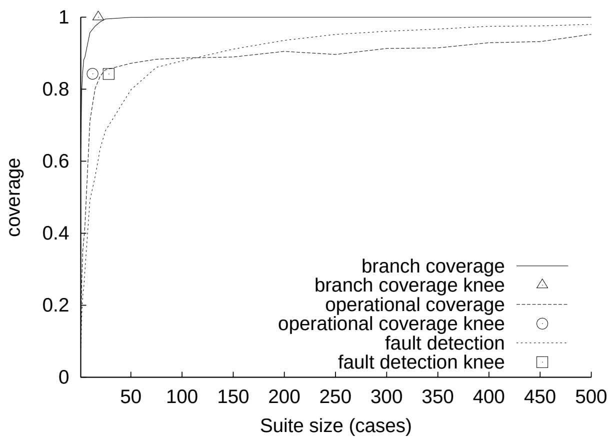

Figure 8. Effect of test suite size (measured in cases) on branch coverage, operational coverage, and fault detection of randomly-generated test suites, for the `replace` program. The other programs have similar graphs, and graphs of fault detection against size in terms of calls are also similar.

against an oracle operational abstraction. Section 6.2 shows that when tests are added at random to a suite, the suite's operational coverage increases rapidly, then levels off at a high value. Section 6.3 shows that operational coverage is correlated with fault detection, even when test suite size and code coverage are held constant.

6.1 Operational coverage

Operational coverage (defined below) measures the difference between an operational abstraction and an oracle or goal specification. Like other coverage measures, operational coverage is a value between 0 and 1 inclusive: 0 for an empty test suite, and 1 for an ideal test suite. It accounts for both false assertions present in, and true assertions missing from, the operational abstraction.

An operational abstraction is a set of assertions about a program, chosen from some grammar. Assume there exists an oracle (goal) operational abstraction G, containing all true assertions in the grammar. We use the oracle only to evaluate and justify the operational difference technique. Application of the OD technique does not require existence of an oracle, though the technique could be extended to take advantage of an oracle when present.

Given a test suite, an operational abstraction OA contains the assertions that are likely true, based on observations made while running the program over the test suite. Define $t = |OA \cap G|$, the number of true assertions in OA. The precision p of OA is the fraction of assertions in OA that are true, so $p = t/|OA|$. The recall r of OA is the fraction of assertions in G that are also in OA, so $r = t/|G|$. Precision and recall are standard measures from information retrieval [Sal68, vR79].

We define the operational coverage c of OA 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 operational abstraction contains 12 assertions: 9 true and 3 false. The precision is $9/12$ and the recall is $9/10$, so the operational coverage is $(9/12 + 9/10)/2 = .825$.

Where do the oracles used in Sections 6.2 and 6.3 come from? The oracle is the set of all invariants in Daikon's grammar that are true. This is exactly the operational abstraction Daikon would generate, given a good enough test suite. In the extreme, the (infinitely large) test suite containing all valid inputs to the program would surely be good enough. We did not have such test suites, so we ran Daikon on the full pool of test cases for each program (Figure 3) as an approximation. We believe that the pools are sufficiently large and diverse that adding additional test cases would change the operational abstraction little or not at all.

6.2 Effect of test suite size on fault detection

When tests are added at random to a suite, the operational 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 operational abstractions are measurably improved by almost any augmentation. Suites inducing good operational abstractions are little affected by augmentation, and the operational abstractions are already so close to ideal that they can never be substantially improved.

Figure 8 plots average branch coverage, operational coverage, and fault detection against suite size for the `replace` program, for 1500 randomly generated test suites of case sizes 1–500. Figure 8 does not plot statement coverage because it lies almost exactly under branch coverage.

Figure 8 also plots the knee of each curve. We computed the knee by finding the size 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. The knee does not necessarily lie on

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

Figure 9. Table of knee locations, averaged across seven programs. These numbers indicate where plots of statement coverage, operational 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 8. The table gives knees using two different metrics for test suite size: number of cases and number of calls.

the curve. Figure 9 gives the average positions of all the knees across all programs. Intuitively, the knee appears at the size that separates low-coverage, easily-improved suites from high-coverage suites that benefit little from the addition of test cases.

The knees are significant for two reasons. First, they indicate roughly what size and coverage users can expect of the generated suites; it is encouraging that they have small sizes and high coverages. More significantly, if the knee was not at a high coverage value, then the slope after the knee would not be very horizontal, and the generation process would take longer to terminate.

6.3 Operational coverage and fault detection

Section 6.2 demonstrated that high absolute levels of operational coverage are achievable. This section shows that increasing the operational coverage (that is, improving the operational abstraction) results in greater fault detection. Section 6.3.1 demonstrates the result for arbitrary test suites, and Section 6.3.2 demonstrates that even when a test suite achieves 100% code coverage, increasing operational coverage improves fault detection.

6.3.1 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 6.2). 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, operational coverage, and fault detection. Then, we performed six multiple regressions for each program. These regressions indicate 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.

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

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

Figure 10. 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 of each table presents the results from a separate multiple regression.

Each column of Figure 10 presents results of one multiple regression. For instance, the upper-left regression uses size and statement coverage as the independent variables, and uses operational 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. This means the coefficients for statement and branch coverage cannot be directly compared. There is no significant interaction effect among any other predictor variables at the $p = .10$ level.)

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

The standardized coefficients in Figure 10 indicate the direction and relative magnitude of correlation between the independent and dependent variables. Test suite runtime is the most important predictor of fault detection, followed by branch coverage, number of cases, statement coverage, and operational coverage.

The important conclusion to draw from this experiment is that operational coverage is a useful predictor of fault detection: test suites with more operational coverage detect faults better. Operational coverage is about as good a predictor of fault detection as statement coverage, but is less good than some other metrics, such as test suite size.

6.3.2 The effect of 100% code coverage

A final experiment further demonstrates the value of operational coverage as a test suite quality metric that is indepen-

Coverage type	Op. cov. coefficient	Mean op. 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 11. Multiple regression coefficient for operational 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 operational 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 operational coverage coefficient was statistically significant.

dent 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 5 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, operational 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 operational coverage as the independent variables and fault detection as the dependent variable. We performed 22 multiple regressions in total (7 programs × 3 coverage criteria, plus 1 coverage criterion for space). Figure 11 summarizes the results.

The coefficient describes the relationship between operational coverage and fault detection. For example, the coefficient of .48 for statement coverage suites suggests that if the operational 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 operational 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 operational coverage does increase fault detection. However, for suites with def-use coverage, fault detection is often independent of operational coverage (only 2 programs had statistically significant coefficients). This might be because operational coverage is already near perfect (.95) for those test suites: there is little room for improvement.

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

6.3.3 Effort to improve test suites

We have demonstrated that both code coverage and operational coverage are correlated to fault detection, and that improving operational coverage tends to detect different faults than improving code coverage does (Section 5.1.1). However, this does not indicate how much relative effort a software tester should invest in improving the two types of coverage. Future work should assess how difficult it is for the programmer to increase operational 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 operational coverage similar?

7 Related work

This work complements and extends research in specification-based testing. For the most part, previous research has focused on systematic generation, not evaluation, of test suites, and has required users to provide a specification *a priori*. We relax those constraints, provide new applications, and show how to bring new benefits to specification-based testing.

7.1 Specifications for test suite generation

Goodenough and Gerhart [GG75b, GG75a] suggest that users partition the input domain into equivalence classes and select 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; our results reinforce this point.

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. Although 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. He 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 operational difference technique assumes the existence of a test case generator (any of the above could serve), then generates both a test suite and an operational abstraction.

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 operational difference technique does not, but provides an operational abstraction.

7.3 Related coverage criteria

Several researchers have proposed specification-based notions of coverage that are similar to our definition of operational 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 operational coverage compares two operational abstractions.

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. Operational 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.

8 Conclusion

We have presented the operational difference (OD) technique for generating, augmenting, and minimizing test suites. The technique selects test cases by comparing operational abstractions dynamically generated from test suites. An operational abstraction describes observed behavior and is syntactically identical to a formal specification. A test case is considered interesting if its addition or removal changes the operational abstraction. The OD technique is automatic, but (for generation or augmentation) assumes the existence of a test case generator that provides candidate test cases.

The OD generation technique produced test suites that are smaller, and slightly more effective at fault detection, than branch coverage suites. The OD suites are equally good at detecting errors that change the control flow graph and are more effective at detecting non-CFG-changing errors. The two techniques are complementary: combining structural and operational coverage techniques is more effective than using either in isolation.

Two characteristics of our technique for inducing operational abstractions explain the efficacy of the OD technique. First, it is possible to create a high-quality operational abstraction (without knowing precisely how good it is or what the ideal would be) by randomly adding test cases until the operational abstraction stabilizes. Second, improvements in the operational abstraction are correlated with fault detection, even for test suites with 100% code coverage. Thus, even after statement or branch coverage can no longer differentiate among test cases, the OD technique can determine which test cases are most advantageous.

The results do not depend on the generated operational abstraction — which describes actual, not intended, behavior — being close to the formal specification that a human might write (though in practice, they are). Likewise, the results do not depend on the program being correct: the OD technique produces test suites with good fault detection even from buggy programs. (In our experiments, the test programs used to generate the operational abstractions contained errors, and further experiments have confirmed that the OD technique is not hindered by errors in the program from which the operational abstraction is induced.) Finally, there is no need for an oracle that determines whether a test case passes or fails, though subsequent use of the generated test suites might require creation of such an oracle.

The operational difference technique is automatic; it takes as input a test case generator (any test suite or sample inputs will do) but does not require a programmer to provide an *a priori* formal specification. Since such specifications are rare and programmers are loathe to write them (because programmers perceive the cost as too high and the benefit as too low), the OD technique expands the applicability of specification-based testing. The technique has the added benefit of generating a high-quality operational abstraction for the program, which has many benefits in itself; however, users not interested in this benefit never need examine the operational abstraction.

The OD technique is directly analogous to techniques for creating code coverage suites, but it operates over program properties in the semantic domain rather than over source code constructs in the syntactic domain. We believe, and our experiments validate, that operating over (automatically generated) program properties rather than program text holds substantial promise for testing and other software engineering tasks.

Acknowledgments

Benjamin Morse wrote the C/C++ front end to the Daikon system and assisted with data collection. We also thank the other members of the Daikon project, most notably Jeremy Nimmer, for their assistance with our research and tools. Gregg Rothermel provided us the Siemens programs and answered questions. Vibha Sazawal provided statistical consulting. This research was supported by NTT, Raytheon, NSF CAREER grant CCR-0133580, and a gift from Edison Design Group.

References

[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.

[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, December 1989.

[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 2000, 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.

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

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

[Har02] Michael Harder. Improving test suites via generated specifications. Technical Report 848, MIT Laboratory for Computer Science, Cambridge, MA, June 4, 2002. Revision of author's Master's thesis.

[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.

[JH01] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 92–101, Florence, Italy, November 6–10, 2001.

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

[NE02a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[NWEG$^+$03] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 283–297, New York, New York, January 9–11, 2003.

[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.

[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.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[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 *Proceedings of the International Conference on Software Maintenance*, pages 44–53, 1998.

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

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.