Specification Coverage as a Measure of Test Suite Quality

Michael Harder Benjamin Morse Michael D. Ernst

MIT Lab for Computer Science
545 Technology Square
Cambridge, MA 02139 USA
{mharder,scruffy,mernst}@lcs.mit.edu
http://sdg.lcs.mit.edu/daikon/

Abstract

This paper introduces specification coverage, a measure of test suite quality that indicates how much of a program's behavior is exercised by a set of executions. Specification coverage complements code-based coverage metrics and may be better for certain tasks. Specification coverage is also complementary to specification-based test suite generation: it can be computed for any test suite and can be used without *a priori* knowledge of the program's specification.

This paper presents results from experiments that derive relationships between test suite size, specification coverage, code coverage (of various sorts), and bug detection. Relatively small test suites achieve high levels of specification coverage. Specification coverage is correlated with bug detection (when controlling for test suite size and code coverage). Specification coverage is correlated with code coverage (when controlling for test suite size). Finally, the paper discusses techniques for automatically producing specification-coverage-complete test suites.

1. Introduction

This paper proposes a new notion of specification coverage that measures specifications induced from program executions. Previous work on specification based testing has concentrated on using a specification to *generate* a test suite. In contrast, we concentrate on using a specification to *evaluate* a test suite. In addition, we provide an experimental evaluation of our technique.

We show that specification coverage is a practical measure of test suite quality. It is possible for a test suite to have near-perfect specification coverage. Even small test suites can have good specification coverage.

We show that specification coverage is a valuable measure of test suite quality. It is a good predictor of bug detection, even when controlling for size and code coverage. As a consequence, specification coverage can be used to improve the quality of test suites with complete code coverage.

Specification coverage can be leveraged for practical programming tasks. A small test suite can be improved by adding cases that improve its specification coverage. A large test suite can be safely shrunk by removing tests that don't contribute to specification coverage.

Specification based testing has not seen widespread use, due in part to lack of tool support. Previous techniques have required a specification, which is rarely present in production code. Our technique does not require a specification. We assume a specification for the purpose of evaluating our results in this paper, but practical uses do not need one.

Without a goal specification, it is not possible to measure the absolute coverage of a suite. It is possible, though, to measure the relative difference in coverage between two suites, by comparing the specifications generated by each suite. We use the Daikon invariant detector to generate a specification from a test suite.

Section 2 introduces the concept of specification coverage and explains the process of specification-based testing. Section 3 outlines our technique for automatically generating a specification, so that specification-based testing can be carried out on a program without a formal specification. We then give a formal definition of specification coverage as a metric. Section 4 contains experimental methodology and overview, and Sections 5–7 describe the experimental results. Section 8 discusses the results and implications of these experiments. Section 9 discusses other work in this area, and other notions of coverage proposed by other researchers that are akin to our definition of specification coverage. Section 10 describes ways in which our research can be extended, and addresses the question of how to create test suites that have the quality of specification coverage.

2. Specification-based testing

Program specifications are useful in many aspects of program development, including design, coding, formal verification, testing, 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 abstract away irrelevant details about program implementation while preserving important properties about the behavior. In addition to making those properties more apparent, this permits reasoning about them by humans or machines.

Specifications play a valuable role in dynamic analyses such as software testing [GG75b, ROT89, CRS96, OL99]. Whereas structural (implementation-based) testing exam-

ines the actual functionality and structure of the implementation, specification-based testing considers intended behavior. Specification-based testing can complement structural testing. For example, it can detect errors (such as missing paths and omitted functionality) that may not be detected by implementation-based testing. Specification-based testing can be performed independently of the details of any particular implementation. The specification is at a higher level than the code, so software engineers may notice problems with requirements, design, or functionality that would otherwise be detected much later or overlooked entirely. Specifications can also be used as an oracle to check test results, and can direct selection of test cases.

This paper proposes and evaluates *specification coverage*, a metric of test suite quality that measures how much of a program's specification is exercised while executing a program on the test suite. Coverage is used in two distinct but related ways in testing: to evaluate test suites and to generate test suites. To date, specifications have been primarily used for test suite generation, and users had to provide the specifications. This paper shows how to use specifications for evaluation, without the need to provide them *a priori*.

Any coverage criterion defines a set of events or conditions. For example, for statement coverage, each event is the execution of a particular line of code, and the complete set includes all such events. A test suite covers the criterion if execution of the suite causes all the events to occur or all the conditions to become true. A test suite can be evaluated by computing its coverage, which is the percentage of the conditions in the set that are satisfied. Any test suite, no matter how it was constructed, can be evaluated according to any criterion.

A test suite can be generated so as to cover a specific criterion. The criterion may be formally stated and measurable, or it may be implicit and informal, in which case the notion of coverage is meant to be evocative of the intuition above: more coverage gives a greater likelihood of detecting errors. If test suite generation is systematic, then it guarantees coverage of the criterion, and there is no need to evaluate the resulting test suite. Another common approach is to repeatedly add tests to a suite and measure coverage (perhaps choosing the tests to add according to the coverage results of the previous round) until the suite covers the criterion.

To date, specification-based test generation has typically not been evaluated, in part because the criterion is not operationalized and in part because tests are systematically generated. This lack of evaluability limits the applicability of specifications to testing and other dynamic analyses. Just as specification-based test suite generation complements other generation strategies, specification-based test suite evaluation can complement other evaluation strategies.

3. Automatically generating specifications

Lack of specifications is a serious obstacle to specification-based test suite evaluation. Very few programs are formally specified, and often even `assert` statements and documentation are absent. Time pressures may prevent programmers from writing such documentation; few programmers enjoy the task of writing specifications or are even trained to do so; specifications can be as large and difficult to write as the original program (85% as large, in one evaluation of specification-based testing [CR99]); and relatively small measurable benefits make writing specifications cost-ineffective in many domains.

Our technique automatically generates and evaluates a specification capable of being used to measure test suite quality. This both lowers the cost of generating specifications and increases the utility of those specifications.

To generate the specification, we use the Daikon system for dynamically detecting likely program invariants [ECGN01, Ern00]. 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. A set of invariants is a lightweight, incomplete, but useful form of specification.

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 system reports properties that hold over execution of an entire test suite (which is provided by the user).

The potential invariants are generated by instantiating, at each procedure entry and exit, each of a set of invariant templates. The templates are filled in with each possible subset of variables that are in scope at the program point, plus certain expressions over those variables. Although there are many potential invariants, testing is efficient because most potential invariants are falsified quickly and need not be tested thereafter.

The output of the invariant detector is improved by suppressing invariants that are not statistically justified, that involve variables that can be statically proved to be unrelated, or that satisfy certain other conditions [ECGN00]. As an example of a statistical test, $\mathtt{x} \neq \mathtt{y}$ should not be reported on the basis of just a few executions, but may be justified if `x` and `y` are never equal over many executions. As another example, every variable `x` takes on a maximum value on any set of executions. The invariant $\mathtt{x} \leq 22$ should not be reported if `x` takes on the value 22 just once, but may be reported if that maximal value is encountered repeatedly. Each variety of invariant has an associated confidence (essentially, a test against the null hypothesis that the invariant does not hold); when this confidence exceeds a user-settable parameter, the invariant may be reported.

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, which can then be corrected. 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://sdg.lcs.mit.edu/daikon/`. While our our experiments rely on the Daikon tool, the ideas generalize beyond any particular implementation of specification generation.

3.1 Comparing specifications

Given a coverage criterion (a set of conditions to satisfy), a test suite's coverage is a ratio between 0 and 1 — the fraction of the conditions that are satisfied by execution of the suite. This measurement requires that the set of conditions

be known in advance.

We measure specification coverage slightly differently, because our technique generates the coverage conditions (the specification) as well as checking coverage; indeed, the approach could be called specification generation coverage. As described in Section 3, each invariant (each part of the specification) is reported only if there is adequate statistical evidence for it [ECGN00]. Some conditions that hold over the entire test suite are not reported if they are insufficiently supported by the program's executions. A single test case may not guarantee coverage of any particular invariant, and it is even possible for adding a test case to degrade the result.

Because our approach generates as well as checks specifications, the set of invariants induced by a test suite may not only omit invariants that appear in the goal set, but may also contain extra (undesired) invariants that do not appear in the goal set. Thus, a simple ratio of detected to goal invariants is not an appropriate measure of specification accuracy. We use the following formulas instead.

$$\begin{aligned} \text{precision} &= \frac{\text{correct}}{\text{reported}} \\ \text{recall} &= \frac{\text{correct}}{\text{goal}} \\ \text{coverage} &= \frac{\text{precision} + \text{recall}}{2} \end{aligned}$$

In these formulas, "goal" is the number of invariants in the goal set, "reported" is the number of invariants induced by the test suite, and "correct" is the number of reported invariants that are also in the goal set. Precision is a standard measure of correctness, and recall is a standard measure of completeness (recall corresponds to ordinary test suite coverage). Specification coverage is the average of precision and recall. Like other coverage measures, it is a value between 0 and 1 inclusive, is 0 for an empty test suite, and is 1 for an ideal test suite.

A separate issue is the need for goal invariants to compare against. Our technique works regardless of how the goal set is selected — for instance, by hand, by static analysis, or by dynamic analysis. While we use a specification derived from a goal set of invariants to evaluate the techniques proposed in this paper, users need not have a perfect specification, or any at all, in order to apply the techniques in practice. (The results of Section 5 suggest that users can easily produce a nearly-perfect specification, with respect to the particular set of invariants that the tools generate.)

For our purposes, a program's "perfect" specification is all the invariants in Daikon's grammar that are true. The goal invariants are exactly what the Daikon invariant detector would output if it were given a good enough test suite for the program. (In the extreme, the test suite containing all valid inputs to the program would surely be good enough.) The programs in our experimental evaluation have sufficiently good test suites: we believe that adding additional tests would not change the specifications appreciably and might not change them at all. We use the specification produced by Daikon on the full test suite as the goal specification.

4. Evaluation overview

This section outlines our experiments to evaluate specification coverage. We related four measures over each test suite: size, code coverage, specification coverage, and bug detection. Figure 1 summarizes the experiments. The target programs and test suites are described in Section 4.2.

Section	Independent variables	Dependent variables	Notes
§5	size	spec. cov. stmt. cov. bug detect.	
§6	size spec. cov. stmt. cov.	spec. cov. stmt. cov. bug detect.	
§7	size spec. cov.	bug detect.	suites with code coverage

Figure 1: Summary of experiments. For each row of the table, the listed section reports how (all combinations of) the independent variables affect the dependent variables. The variables are size (test cases and function calls), statement coverage, specification coverage, and bug detection. Section 7 treats each of statement, edge, and def-use coverage as a binary quantity.

Section 5 measures how specification coverage, statement coverage, and bug detection increase as test suite size increases for random test suites. These measurements relate specification coverage to other frequently measured quantities and permit comparing our programs and test suites to those used in other research. Additionally, these measurements indicate what test suite size ranges are sensible for further measurements.

Section 6 measures how the same factors — specification coverage, statement coverage, and bug detection — are affected by changes in test suite size, specification coverage, and statement coverage. We performed a multiple regression of all three predictors against each criterion variable. This regression indicates how each predictor affected each result, while holding all other factors constant; for example, it avoids conflating the effect of size and coverage, even though (as shown in Section 5) larger suites tend to have more coverage. (There is no significant interaction effect among the predictor variables at the $p = .10$ level.)

Section 7 measures similar effects as Section 6, using the same multiple regression technique, but only considering test suites with complete code coverage. Rather than measuring coverage as a fraction of conditions satisfied, it only considers whether a suite has complete coverage. Such measurements are interesting because testers are frequently counseled to achieve or attempt complete coverage; Section 7 reveals properties of such suites.

4.1 Measurement details

This section describes how each of the measurements — test suite size, statement coverage, specification coverage, and bug detection — was performed.

We have two versions of every program: the original version and a slightly modified version (see Section 4.2.2) that can be instrumented by the Daikon front end for C. The instrumented version behaves identically to the uninstrumented version, except that it writes trace data to a file and checks for memory errors (and runs slightly slower as a result). We measured statement coverage and bug detection using the original program. We measured specification coverage, number of calls executed, and lines of code using the

modified program.

Lines of Code. We measured the total lines of code in the programs, after we modified them as described in Section 4.2.2. We used the Unix `wc` command to count the number of lines in the C source file and any non-system header files it included.

To compute non-comment non-blank lines of code, we removed all comments from the programs using a Perl script, then removed all blank lines with the Unix `grep` command. We used the Unix `wc` command to count the number of lines in the resulting files.

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. We measured the number of calls by counting the number of function entry lines in Daikon's data trace file [Ern00].

Code coverage. We considered three different varieties of code coverage: statement coverage, branch coverage, and definition-use (def-use or du) coverage. The experiments of Section 7 use suites generated so as to achieve 100% coverage on each of the three criteria. (See Section 4.2 for an explanation and a caveat.)

We also measured statement coverage for each suite using the GCC `gcov` tool. Unreachable code in the programs prevents `gcov` from ever reporting 100% statement coverage. We normalized all statement coverage measurements by dividing by the number of reachable statements, which we computed by running `gcov` on a pool of tests designed to cover all reachable statements (see Figure 2).

We did not measure branch or def-use coverage. (We could not use `gcov` to measure branch coverage, because `gcov` calculates branch coverage of the assembly code, not the source code.)

Specification coverage. We measured specification coverage via the procedure of Section 3.1. The goal specification is the set of invariants produced by Daikon using the entire test pool. Figure 2 gives the sizes of these specifications.

Bug 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 bug detection rate of a test suite is the ratio of the number of faulty versions detected to the number of faulty program versions. Section 4.2 describes how faulty versions were selected.

4.2 Subject programs

Our experiments analyze seven C programs that were created by Siemens Research [HFGO94] and subsequently modified by Rothermel and Harrold [RH98]. The programs come with test suites and faulty versions.

Each program is associated with a pool of tests. 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. Figure 2 shows the size of the test pools.

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.) Figure 2 indicates how many faulty versions remained.

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 were 1000 test suites for each type of coverage.

4.2.1 Problems with the coverage suites

Not all of the statement coverage suites actually achieved 100% statement coverage as measured by `gcov`, even when normalized by the universe. We verified this behavior by adding abort statements at certain locations that were triggered by some statement coverage suites but not by others. When normalized by the universe, the mean statement coverage of the statement coverage suites was greater than 99% for each program. The minimum statement coverage by a statement coverage suite was 90% (for a `tcas` test suite). 86% of the statement coverage suites actually had statement coverage.

Additionally, we found array-overrun bugs in three of the programs (`print_tokens2`, `replace`, and `tcas`). These errors had not been noticed in previous research using the programs. When the coverage test suites were originally created, the erroneous programs had read or written an element beyond the bounds of an array without inducing a fault. However, in our environment, the array bounds errors caused the programs to crash. (Actually, they corrupted the Daikon runtime's data structures and triggered an assertion failure, so we added runtime checks for such memory errors to the Daikon runtime.) This program crash is legal behavior in C: accessing beyond the bounds of an array may cause unspecified behavior, including the program continuing without error, crashing, or changing the value of unrelated variables. As a result of the crashes, coverage may be lower in our environment than that in which the tests were generated. If a test case covered a condition only after an array overrun when the test suites were generated, then the condition would not be covered when the case is run in our environment.

To determine the severity of this second problem, we ascertained the branch coverage in our environment of the "branch coverage" suites for each of the 3 programs. For `print_tokens2`, 6 out of 1000 test suites failed to achieve coverage. For `replace`, 900 of 1000 test suites failed to

Program	Program size			Faulty	Test pool				Description
	Functions	LOC	NCNB	versions	cases	calls	stmt. cov.	spec. size	of program
`print_tokens`	18	703	452	7	4130	619424	.931	97	lexical analyzer
`print_tokens2`	19	549	379	10	4115	723937	.980	173	lexical analyzer
`replace`	21	506	456	32	5542	1163810	.953	252	pattern replacement
`schedule`	18	394	276	9	2650	442175	.977	283	priority scheduler
`schedule2`	16	363	280	9	2710	954221	.968	161	priority scheduler
`tcas`	9	175	136	41	1608	12613	.973	328	altitude separation
`tot_info`	7	556	334	23	1052	13208	.952	156	information measure
Average	15	464	330	19	3115	561341	.962	207	

Figure 2: 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. Statement coverage is percentage of total statements, including unreachable statements; all subsequently reported statement coverage values are normalized by dividing by the numbers in this column. Specification size is the number of invariants generated by the Daikon invariant detector when run over the program and the full test pool.

achieve coverage. For `tcas`, 73 of 1000 test suites failed to achieve coverage.

Unfortunately, we did not have the tools or resources to generate new suites that would provide coverage in our environment. Thus, we used the test suites provided to us, with the knowledge that some of the claims made about the suites cannot be relied upon.

4.2.2 Changes to the programs

This section lists the changes we made to the programs we were provided.

The Daikon front end for C accepts as input the subset of ANSI C that is also C++, and it produces C++ as output. Since the Siemens programs are K&R C, we converted them to ANSI C. First, we processed each program with the GCC utility `protoize`. Then, we removed typedefs for `bool`, which is a reserved word in C++, from the `replace`, `tcas`, and `tot_info` programs. We also removed definitions of NULL that conflicted with the C++ system include files, and we renamed one variable that had the same name as a type.

To work around a limitation of the Daikon front end for C, we added forward declarations of each global variable at the beginning of each program. The original programs declared global variables at various points throughout each program.

We reintroduced faults into three "faulty" versions of `replace` that were provided to us in a corrected state. Those faults had been corrected because previous research used tools that could not process certain source code structures.

We fixed a typographical error in `print_tokens2` where a parameter variable to the procedure `unget_error` was mistakenly declared as the wrong type. The static typechecking in C++ would not allow us to compile until the error was remedied.

We eliminated one faulty version (version 9 for `schedule2`) that was not detected by its test pool; a hand analysis revealed that the faulty version could never behave differently than the original version.

The values we calculated for lines of code in Figure 2 do not agree with previously reported numbers [RH98]. We were unable to reproduce those results, which were presented without explanation of how they were computed.

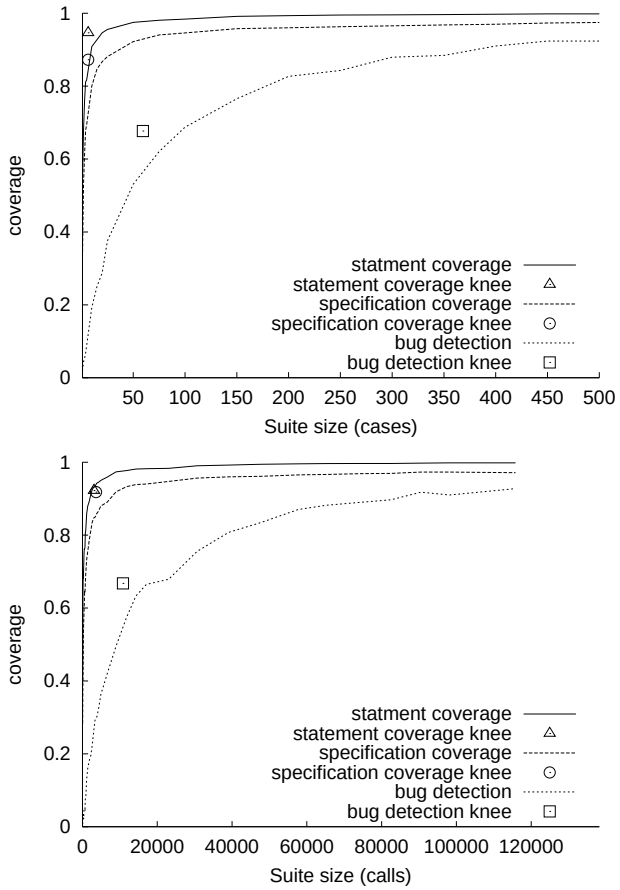

Figure 3: Effect of test suite size (measured in both test cases and function calls) on statement coverage, specification coverage, and bug detection of randomly-generated test suites. These data are for the `replace` program, but plots for the other programs had similar shapes.

5. Effect of test suite size

	cases knee		calls knee	
	cases	value	calls	value
stmt. cov.	8	0.97	873	0.92
spec. cov.	11	0.85	1222	0.86
bug detection	60	0.74	10655	0.74

Figure 4: Table of knee locations, averaged across seven programs. These numbers indicate where plots of statement coverage, specification coverage, and bug detection against time switch from one nearly-linear component to another; they indicate average positions of the knees plotted for one program in Figure 3. 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.

Our first investigation measured the specification coverage, statement coverage, and bug detection of randomly generated test suites. This experiment verifies that specification coverage is achievable and gives an idea of how large specification coverage suites might be.

For each program, we generated a total of 1475 test suites of case sizes 1–500 (with more suites of smaller sizes, where coverage measurements change most rapidly), picked randomly from the test pool. Larger test suites are not necessarily supersets of the smaller suites. The suites had cases sizes of 1–5, 10, 15, and 20 (100 suites each); 25, 50, and 75 (75 cases each); and each multiple of 50 from 100 to 500 (50 suites each).

Figure 3 plots average statement coverage, specification coverage, and bug detection for each suite size, for the `replace` program from the Siemens suite.

In the plots for all seven programs, all measured values rise sharply as suite size increases, then level off to a nearly linear approach towards the asymptote. Figure 3 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 4 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 8 tests. Specification coverage levels off only slightly later, but at a lower coverage value. Bug detection takes the longest to reach its knee, and even thereafter continues to rise appreciably even as test suites are made very large.

These results differ from those of a previous paper, which suggested that specification coverage would reach an asymptote between 500 and 1000 test cases and that invariant sets would contain thousands of elements [ECGN01]. Our results show that the knee occurs at a much smaller size of suite and output. The differences are explained by a new implementation of dynamic invariant detection. The most significant change is a collection of relevance improvements [ECGN00] that eliminate output that is not statistically justified, is implied by other output, or relates incomparable values. For example, the Daikon front end for C now uses the Lackwit static analysis tool [OJ97] to determine which variables should be compared to one another. One other change is that (as of the time of these experiments) the Daikon front end for C does not output fields of `structs`, so invariants over them do not appear.

Finding the shape of the coverage curves is important for multiple reasons. Firstly, for our benefit, it allows us to go on with the rest of the experiments in this paper. Knowing that specification coverage is achievable in general, and that it increases with the size of a randomly generated test suite, makes measurements of it valid. In addition, knowing that there is a knee below which specification coverage has a wide range gives us an idea of the domain of test suite sizes in which to study changes in specification coverage.

Discovery of a sharp, evident knee in the specification coverage curve indicates what size random test suite will achieve good specification coverage. We can then iteratively increase the size of a random test suite and look for the knee. Knowledge of the relative positions of the specification coverage, statement coverage, and bug detection knees is quite useful — since the specification coverage knee appears at roughly the same size random test suite as the statement coverage knee, one can iteratively add random cases to a test suite until statement coverage is achieved, and be assured of moderately good specification coverage.

Continuing to grow this test suite to 7.5 times its size when it first reached specification coverage suggests that good bug detection can be achieved, as 7.5 is the ratio of the location of the bug detection knee to the statement coverage knee.

Knowing that the slope of the specification coverage curve changes sharply after the knee suggests a better and more reliable method for generating specifications: iteratively add to a test suite until the changes in the invariants slow down noticeably. At that point, the specification coverage knee has been reached.

The height of the specification coverage knee demonstrates a valuable property of specification coverage. Though the inflection point is reached early, similarly to statement coverage, there's still room for improvement. Incremental improvements in specification coverage can still be attained, even when statement coverage has reached its maximum.

6. Correlation between specification coverage and bug detection

Software testers want their test suites to detect as many bugs as possible. Since it is impossible to directly predict how many bugs a suite will detect, other measures are used to predict bug detection, such as code coverage. This experiment shows that specification coverage is as good a predictor of bug detection as code coverage. Holding all other factors constant, test suites with more specification coverage detect more bugs.

We analyzed 1000 test suites for each of the seven programs. The suite sizes, in cases, were uniformly distributed between 1 and the number of cases at the bug detection knee (Section 5). The cases in each suite were chosen at random.

For each test suite, we calculated its size, statement coverage, specification coverage, and bug detection, as previously described. Then, we performed three multiple regressions for each program. The first regression uses size and statement coverage as the independent variables, and uses specification coverage as the dependent variable. The second regression uses size and specification coverage as the independent variables, and uses statement coverage as the dependent variable. The third regression uses size, statement coverage, and specification coverage as the independent variables, and bug detection as the dependent variable.

Independent variable	Dependent variable		
	spec. cov.	stmt. cov.	bug detect.
cases	.00200	.00038	.00309
calls	.00007	−.00002	.00027
calls-adjusted	.00057	−.00004	.00531
spec. cov.	-	.444	.340
stmt. cov.	1.057	-	.378

Figure 5: Multiple regression coefficients, averaged across seven programs. The coefficients can be compared across a row, but not down a column. The parameters for cases and calls adjusted can be compared, as can the parameters for specification coverage and statement coverage. Each column presents the results from a separate multiple regression.

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

Figure 6: Standardized multiple regression coefficients, averaged across seven 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.

The results are presented in Figure 5 and Figure 6.

The coefficients indicate the direction and magnitude of correlation between the independent and dependent variables. For example, when specification coverage is used to predict bug detection, the coefficient is .340. This means that if specification coverage is increased by 1 percent, and all other predictors are held constant, bug detection increases by .340 percent.

The coefficient for "calls-adjusted" has been multiplied by the average number of calls per case before performing the regression. This makes the coefficients for cases and calls-adjusted comparable. The coefficient for calls adjusted indicates how much the dependent variable will change if you add a test case with an average number of calls for that program. It indicates whether more of the benefit from such a test case comes from the case itself or from the calls performed by the case. (For coverage, most of the benefit comes from the case, but for bug detection, two thirds of the benefit comes from the calls.)

The coefficients can be compared across a row, but in general cannot be compared down a column. The coefficients for cases and calls adjusted are comparable, as are the coefficients for specification coverage and statement coverage. However, the coefficients cannot be compared any other way. For example, the coefficients for cases and specification coverage cannot be compared.

We analyzed the results by examining each dependent variable separately.

Specification Coverage. Statement 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 calls have small positive effects on statement coverage.

Statement Coverage. Specification coverage has a substantial effect on statement coverage. If a specification is covered well, every statement 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.

Bug Detection. Specification coverage and statement coverage have approximately the same effect on bug detection. Holding all other predictors constant, an increase in statement coverage has about the same effect as an identical increase in specification coverage. Cases and calls both have a positive effect on bug detection.

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

	statement		branch		def-use	
Program	cases	calls	cases	calls	cases	calls
`print_tokens`	15.1	6369	16.2	6546	38.4	17856
`print_tokens2`	11.8	2661	11.9	2686	34.9	8905
`replace`	17.9	4088	18.7	4340	64.4	23528
`schedule`	5.9	751	8.3	1100	23.9	3294
`schedule2`	7.1	1989	7.9	2215	25.8	6758
`tcas`	5.1	48	5.8	50	5.6	48
`tot_info`	6.8	109	7.2	115	15.0	235
Average	10.0	2288	10.9	2436	29.7	8661

Figure 7: Average number of cases and calls in the coverage suites.

7. The 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 the code coverage metric.

For each program, we analyzed 1000 suites with statement coverage, 1000 suites with branch coverage, and 1000 suites with def-use coverage. Section 4.2 describes these test suites. Figure 7 presents the average number of cases in the test suites. 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 bug 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 bug detection as the dependent variable. We performed 21 multiple regressions in total (7 programs × 3 coverage criteria). Figure 8 summarizes the results.

The coefficient describes the relationship between specification coverage and bug 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 bug detection rate would increase by approximately .48 percent.

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

Coverage type	Spec. cov. coefficient	Mean spec. cov.	Mean bug detect	Number of programs
statement	.483	.877	.396	5
branch	.308	.866	.461	6
def-use	.507	.950	.763	2

Figure 8: Multiple regression coefficient for specification coverage, when regressed against bug 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 last column contains this number. Each value was averaged across all programs for which the specification coverage coefficient was statistically significant.

These results show that, for test suites with branch or statement coverage, increasing specification coverage does increase bug detection. However, for suites with def-use coverage, bug detection is often independent of specification coverage. This might be because specification coverage is already near perfect.

Further, these results show that specification coverage is complimentary to code coverage for detecting bugs. Even when statement or branch coverage is 100%, increasing specification coverage may increase the bug detection of a test suite.

8. Discussion

8.1 Specification-free specification coverage

One unusual aspect of our specification-coverage technique is that it does not require the program's specification to be known *a priori*. Rather, it generates the specification from the test suite. In other words, this is a black-box coverage technique, insofar as it is not assumed that a specification already exists. Other coverage techniques require that the coverage criteria — typically, the structure whose parts should all be touched — be available, because otherwise there is no way to know that the criteria have been satisfied. A specification is a valuable end in itself, because it can be used as documentation and for a variety of software engineering tasks. Thus, generation of the specification is an extra benefit of our coverage technique.

Another difference from other coverage notions is that test cases do not individually guarantee that an invariant is reported; rather, several cases may be required, and adding cases may even reduce the size of the generated specification. (Also see Section 10.1.) This can be seen as a positive or a negative trait: properties are not reported based on a single sample, but only after multiple samples have built confidence. As a result, specification coverage cannot be directly used for test suite minimization, though the resulting specification could be used just like any other specification in structural specification-based test suite evaluation. Specification coverage also might be disproportionately worse for smaller test suites.

8.2 Threats to validity

While the results we have presented are suggestive and promising, they might not generalize to other programs or test suites. This section discusses some characteristics of our experiments that might limit the applicability of the results.

The subject programs are quite small, and larger programs might have different characteristics. However, these programs were used in previous testing research, permitting comparison of our results with previous ones. The main reason to choose them 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 the Siemens programs differ from large programs less than machine-generated tests and faults differ from real ones.

Our results examine test suites of modest sizes, up to the knee of the bug detection curve. Very small or very large test suites probably have different characteristics. However, almost any augmentation of a small test suite is an improvement, and few augmentations of large test suites have a large effect. We tried to examine sizes where we can have an impact and where real-world test suites may lie.

The test suites were designed with code coverage in mind, and the test cases and faults may have been as well. This may mean that this suite is worse for specification coverage, and better for code coverage, than typical test suites. The filtering of faults based on bug detection has an undetermined effect. The small number of faults for certain programs made it hard to correlate bug detection with other measures. The combination of few statements — sometimes only 100 — and high coverage similarly quantized statement coverage.

As noted in Section 4.2, the coverage suites do not always cover the code.

The current set of invariants produced by the Daikon invariant detection tool may be too simplistic. A realistic specification might include more complicated invariants, and specification coverage might be lower (or higher) for such a set.

Despite these potential problems, we believe that the results should be of general applicability in other situations. We intend to evaluate our tools on other programs, test suites, and faulty versions.

9. Related work

This work builds on research in specification-based testing. For the most part, that research has focused on generation, not evaluation, of test suites.

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 then be used as test adequacy metrics.

Offut et al. generate tests from constraints that describe path conditions and erroneous state [Off91] and from SOFL specifications [OL99]. Other researchers have extended work on generating test cases from specifications [DF93, Don97, Meu98].

The research most closely related to ours evaluates test suites based on programmer-supplied specifications. Chang and Richardson [CR99] convert specifications written in ADL [HS94] into a series of checks in the code called coverage condition functions [CRS96]. Once the specification is converted into code, code coverage techniques can be applied

directly: they run the test suite and count how many of the checks are covered.

9.1 Related coverage criteria

Several researchers have proposed notions of coverage that are similar to our definition of specification coverage.

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].

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. Ernst [Ern00] uses the term value coverage to refer to covering all of a variable's values (including boundary values). Hamlet's probable correctness theory [Ham87] calls for uniformly sampling the possible values of all values.

Chang and Richardson's operator coverage [CR99] concerns the creation of mutant (faulty) versions of programs, in order to assess test suite comprehensiveness. Operator coverage is achieved if every operator in the program is changed in some mutant version.

10. Future work

This research was performed on programs with a specification (generated from a universe of all possible tests). In a production environment, a universe test suite is rarely available, so calculating specification coverage by comparing the set of invariants produced with a small test suite to an ideal set of invariants may not be possible. However, we have outlined methods by which someone could iteratively generate a test suite. This technique requires experimental evaluation.

We were unable to relate specification coverage to branch coverage, as the `gcov` tool did not measure the sense of branch coverage that we were interested in. Using a different tool, the branch coverage of the suites we used could be calculated, and then related to specification coverage accordingly.

We also did not measure which specific bugs were detected by each individual suite. It is possible that increasing specification coverage helps to detect a different set of bugs than are detected by increasing statement coverage.

We have demonstrated that both statement coverage and specification coverage have roughly the same correlation to bug detection, but to provide a deeper analysis, it would be interesting to assess how much work it is to the programmer to increase specification coverage, relative to the work it takes to achieve a similar gain in statement coverage. The last few percent of statement coverage are the hardest to achieve; is specification coverage similar?

10.1 Creating specification-covering suites

The utility of specification coverage for testing and the value of simply having an accurate specification make producing specification-covering suits attractive. They cannot be constructed in the same way as code-covering suites for two reasons. First, each test case either satisfies or does not satisfy each code coverage criterion; with specification coverage, many cases may add confidence to an invariant, but no one case may alone justify the invariant. Second, adding a case to a test suite can reduce coverage, for instance by justifying an undesirable invariant.

We have begun preliminary investigations into creating specification-covering suites. We have solved the first problem by making justification vary continuously from 0 to 1 and scoring test suites based on partial satisfaction of a specification coverage criterion. (An alternative solution would be to add cases several at a time.) This change made it possible to create specification-covering suites for all seven programs. However, these suites are very large, averaging 478 cases. We are investigating further improvements to our technique and whether large suites are always necessary for our implementation of specification coverage.

Acknowledgments

We are grateful for the assistance of Gregg Rothermel, who generously provided us the Siemens programs and his test suites for them, re-measured branch coverage at our request, and answered questions. We also thank the other members of the Daikon project, notably Jeremy Nimmer, for their assistance with our research and tools. Steve Wolfman made helpful comments on a draft of this paper, and Vibha Sazawal provided statistical consulting. This research was supported by NTT, Raytheon, an NSF ITR grant, and a gift from Edison Design Group.

References

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

[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 *ESEC/FSE*, pages 285–302, September 6–10, 1999.

[CRS96] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *ISSTA*, pages 62–70, 1996.

[DF93] J. Dick and A. Faivre. Automating the generating and sequencing of test cases from model-based specifications. In *Formal Methods Europe*, pages 268–284, 1993.

[Don97] Michael R. Donat. Automating formal specification-based testing. In *TAPSOFT '97*, 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*, pages 449–458, June 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001. A previous version appeared in *ICSE*, pages 213–224, Los Angeles, CA, USA, May 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.

[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.

[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 *ICSE*, pages 191–200, May 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.

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

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

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

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

[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 TSE*, 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.