EXPERIENCE WITH THE COST OF DIFFERENT COVERAGE GOALS FOR TESTING

Brian Marick
Motorola, Inc.

In coverage-based testing, coverage conditions are generated from the program text. For example, a branch generates two conditions: that it be taken in the false direction and in the true direction. The proportion of coverage conditions a test suite exercises can be used as an estimate of its quality. Some coverage conditions are impossible to exerise, and some are more cost-effectively eliminated by static analysis. The remainder, the feasible coverage conditions, are the subject of this paper.

What percentage of branch, loop, multi-condition, and weak mutation coverage can be expected from thorough unit testing? Seven units from application programs were tested using an extension of traditional black box testing. Nearly 100% feasible coverage was consistently achieved. Except for weak mutation, the additional cost of reaching 100% feasible coverage required only a few percent of the total time. The high cost for weak mutation was due to the time spent identifying impossible coverage conditions.

Because the incremental cost of coverage is low, it is reasonable to set a unit testing goal of 100% for branch, loop, multi-condition, and a subset of weak mutation coverage. However, reaching that goal after measuring coverage is less important than nearly reaching it with the initial black box test suite. A low initial coverage signals a problem in the testing process.

BIOGRAPHICAL SKETCH

Brian Marick graduated from the University of Illinois in 1981 with a BA in English Literature and a BS in Mathematics and Computer Science. Until 1989, he worked in product development as a programmer, tester, and line manager, while attending graduate school as a hobby. The work reported here was a result of a research project funded by a Motorola Partnerships in Research grant through Motorola's Software Research Division. He is currently applying the techniques and tools reported here to Motorola products, training others in their use, and planning further investigations of cost-effective and predictable testing techniques.

Author's address: Motorola, 1101 E. University, Urbana, IL 61801.
Phone: (217) 384-8500
Email: marick@cs.uiuc.edu or marick@urbana.mcd.mot.com.

1. Introduction

One strategy for testing large systems is to test the low-level components ("units") thoroughly before combining them. The expected benefit is that failures will be found early, when they are cheaper to diagnose and correct, and that the cost of later integration or system testing will be reduced. One way of defining "thoroughly" is through coverage measures. This paper addresses these questions:

(1) What types of coverage should be measured?

(2) How much coverage should be expected from black box unit testing?

(3) What should be done if it is not achieved?

This strategy is expensive; in the last section, I discuss ways of reducing its cost. A different strategy is to test units less thoroughly, or not at all, and put more effort into integration and system testing. The results of this study have little relevance for that strategy, though they may provide some evidence in favor of the first strategy.

Note on terminology: "unit" is often defined differently in different organizations. I define a unit to be a single routine or a small group of closely related routines, such as a main routine and several helper routines. Units are normally less than 100 lines of code.

1.1. Coverage

Coverage is measured by instrumenting a program to determine how thoroughly a test suite exercises it. There are two broad classes of coverage measures. Path-based coverage requires the execution of particular components of the program, such as statements, branches, or complete paths. Fault-based coverage requires that the test suite exercise the program in a way that would reveal likely faults.

1.1.1. Path-based Coverage

Branch coverage requires every branch to be executed in both directions. For this code

```
if (arg > 0)
{
  counter = 0;       /* Reinitialize */
}
```

branch coverage requires that the IF's test evaluate to both TRUE and FALSE.

Many kinds of faults may not be detected by branch coverage. Consider a test-at-the-top loop that is intended to sum up the elements of an array:

```
sum = 0;
while (i > 0)
{
  i -= 1;
  sum = pointer[i];            /* Should be += */
}
```

This program will give the wrong answer for any array longer than one. This is an important class of faults: those that are only revealed when loops are iterated more than once. Branch coverage does not force the detection of these faults, since it merely requires that the loop test be TRUE at least once and FALSE at least once. It does not require that the loop ever be executed more than once. Further, it doesn't require that the loop ever be skipped (that is, that i initially be zero or less). *Loop coverage* [Howden78] [Beizer83] requires these two things.

Multi-condition coverage [Myers79] is an extension to branch coverage. In an expression like

```
if (A && B)
```

multi-condition coverage requires that A be TRUE in some test, A be FALSE in some test, B be TRUE in some test, and B be FALSE in some test. Multi-condition coverage is stronger than branch coverage; these

two inputs

A == 1, B == 1
A == 0, B == 1

satisfy branch coverage, but do not satisfy multi-condition coverage.

There are other coverage measures, such as dataflow coverage [Rapps85] [Ntafos84]. They are not measured in this experiment, so they are not discussed here.

1.1.2. Fault-based Coverage

In *weak mutation coverage* [Howden82] [Hamlet77], we suppose that a program contains a particular simple kind of fault. One such fault might be using <= instead of the correct < in an expression like this:

if (A <= B)

Given this program, a weak mutation coverage system would produce a message like

"gcc.c", line 488: operator <= might be <

This message would be produced until the program was executed over a test case such that $(A \leq B)$ has a different value than $(A < B)$. That is, we must satisfy a *coverage condition* that

$$(A \leq B) \neq (A < B)$$

or, equivalently, that

$$A = B$$

Notice the similarity of this requirement to the old testing advice: "always check boundary conditions".

Suppose we execute the program and satisfy this coverage condition. In this case, the incorrect program (the one we're executing) will take the wrong branch. Our hope is that this incorrect program state will persist until the output, at which point we'll see it and say "Bug!". Of course, the effect might not persist. However, there's evidence [Marick90] [Offutt91] that over 90% of such faults will be detected by weak mutation coverage. Of course, not all faults are such simple deviations from the correct program; probably a small minority are [Marick90]. However, the hope of weak mutation testing is that a test suite thorough enough to detect such simple faults will also detect more complicated faults; this is called the *coupling effect* [DeMillo78]. The coupling effect has been shown to hold in some special cases [Offutt89], but more experiments are needed to confirm it in general.

There are two kinds of weak mutation coverage. *Operator* coverage is as already described -- we require test cases that distinguish operators from other operators. *Operand* coverage is similar -- for any operand, such as a variable, we assume that it ought to have been some other one. That is, in

my_function(A)

we require test cases that distinguish A from B (coverage condition $A \neq B$), A from C, A from D, and so on. Since there may be a very large number of possible alternate variables, there are usually a very large number of coverage conditions to satisfy. The hope is that a relatively few test cases will satisfy most of them.

1.2. Feasible Coverage Conditions

All coverage techniques, not just mutation coverage, generate coverage conditions to satisfy. (A branch, for example, generates two conditions: one that the branch must be taken true, and one that it must be taken false.) Ideally, one would like all coverage conditions to be satisfied: 100% coverage. However, a condition may be impossible to satisfy. For example, consider this loop header:

for(i = 0; i < 4; i++)

Two loop conditions cannot be satisfied: that the loop be taken zero times, and that the loop be taken once.

Not all impossible conditions are this obvious. Programmer "sanity checks" also generate them:

```
phys = logical_to_physical(log);
if (NULL_PDEV == phys)
   fatal_error("Program error:  no mapping.");
```

The programmer's assumption is that every logical device has, at this point, a physical device. If it is correct, the branch can never be taken true. Showing the branch is impossible means independently checking this assumption. There's no infallible procedure for doing that.

In some cases, eliminating a coverage condition may not be worth the cost. Consider this code:

```
if (unlikely_error_condition)
  halt_system();
```

Suppose that **halt_system** is known to work and that the unlikely error condition would be tremendously difficult to generate. In this case, a convincing argument that **unlikely_error_condition** indeed corresponds to the actual error condition would probably be sufficient. Static analysis - a correctness argument based solely on the program text - is enough. But suppose the code looked like this:

```
if (unlikely_error_condition)
  recover_and_continue();
```

Error recovery is notoriously error-prone and often fails the simplest tests. Relying on static analysis would usually not be justified, because such analysis is often incorrect -- it makes the same flawed implicit assumptions that the designer did.

Coverage conditions that are possible and worth testing are called *feasible*. Distinguishing these from the others is potentially time-consuming and annoyingly imprecise. A common shortcut is to set goals of around 85% for branch coverage (see, for example, [Su91]) and consider the remaining 15% infeasible by assumption. It is better to examine each branch separately, even if against less deterministic rules.

1.3. Coverage in Context

For the sake of efficiency, testing should be done so that many coverage conditions are satisfied without the tester having to think about them. Since a single black box test can satisfy many coverage conditions, it is most efficient to write and run black box tests first, then add new tests to achieve coverage.

Test design is actually a two stage process, though the two are usually not described separately. In the first stage, *test conditions* are created. A test condition is a requirement that at least one test case must satisfy. Next, test cases are designed. The goal is to satisfy as many conditions with as few test cases as possible [Myers79]. This is partly a matter of cost, since fewer test cases will (usually) require less time, but more a matter of effectiveness: complex test cases are more "challenging" to the program and are more likely to find faults through chance.

After black box tests are designed, written, and run, a coverage tool reports unsatisfied coverage conditions. Those which are feasible are the test conditions used to generate new test cases, test cases that improve the test suite. Our hope is that there will be few of them.

1.4. The Experiment

Production use of a branch coverage tool [Zang91] gave convincing evidence of the usefulness of coverage. Another tool, GCT, was built to investigate other coverage measures. It was developed at the same time as a variant black box testing technique. To tune the tool and technique before putting them to routine use, they were used on arbitrarily selected code from readily available programs. This early experience was somewhat surprising: high coverage was routinely achieved. A more thorough study, with better record-keeping and a stricter definition of feasibility, was started. Consistently high coverage was again seen, together with a low incremental cost for achieving 100% feasible coverage, except for weak mutation coverage. These results suggest that 100% feasible coverage is a reasonable testing goal for unit testing.

Further, the consistency suggests that coverage can be used as a "signal" in the industrial quality control sense [DeVor91]: as a normally constant measure which, when it varies, indicates a potential problem in the testing process.

2. Materials and Methods

2.1. The Programs Tested

Seven units were tested. One was an entire application program. The others were routines or pairs of routines chosen from larger programs in widespread use. They include GNU make, GNU diff, and the RCS revision control system. All are written in C. They are representative of UNIX application programs.

The code tested ranged in size from 30 to 272 lines of code, excluding comments, blank lines, and lines containing only braces. The mean size was 83 lines. Size can also be measured by total number of coverage conditions to be satisfied, which ranged from 176 to 923, mean 479. On average, the different types of coverage made up these percentages of the total:

Branch	Loop	Multi	Operator	Operand
7.5%	3.0%	2.7%	16.1%	70.7%

In the case of the application program, the manual page was used as a specification. In the other cases, there were no specifications, so I wrote them from the code.

The specifications were written in a rigorous format as a list of preconditions (that describe allowable inputs to the program) and postconditions (each of which describes a particular result and the conditions under which it occurs). The format is partially derived from [Perry89]; an example is given in Appendix A.

2.2. The Test Procedure

The programs were tested in five stages. Each stage produced a new test suite that addressed the shortcomings of its predecessors.

2.2.1. Applying a Variant Black Box Technique

The first two test suites were developed using a variant black box testing technique. It is less a new technique than a codification of good practice. Its first stage follows these steps:

Black 1: Test conditions are methodically generated from the *form* of the specification. For example, a precondition of the form "X must be true" generates two test conditions: "X is true" and "X is false", regardless of what X actually is. These test conditions are further refined by processing connectives like AND and OR (using rules similar to cause-effect testing [Myers79]). For example, "X AND Y" generates three test cases:

X is true, Y is true.
X is false, Y is true.
X is true, Y is false.

Black 2: Next, test conditions are generated from the *content* of the specification. Specifications contain *cliches* [Rich90]. A search of a circular list is a typical cliche. Certain data types are also used in a cliched way. For example, the UNIX pathname as a slash-separated string is implicit in many specifications. Cliches are identified by looking at the nouns and verbs in the specification: "search", "list", "pathname".

The implementations of these cliches often contain cliched faults.[1] For example:

[1] There's some empirical evidence for this: the Yale bug catalogues [Johnson83], [Spohrer85] are collections of such cliched faults, but only those made by novice programmers. More compelling is the anecdotal evidence: talk to programmers, describe cliched errors, and watch them nod their heads in recognition.

(1) If the specification includes a search for an element of a circular list, one test condition is that the list does not include the element. The expectation is that the search might go into an infinite loop.

(2) If a function decomposes and reconstructs UNIX pathnames, one test condition is that it be given a pathname of the form "X//Y", because programmers often fail to remember that two slashes are equivalent to one.

Because experienced testers know cliched faults, they use them when generating tests. However, writing the cliches and faults down in a catalog reduces dependency on experience and memory. Such a catalog has been written; sample entries are given in Appendix B.

Black 3: These test conditions are combined into test cases. A test case is a precise description of particular input values and expected results.

The next stage is called broken box testing[2]. It exposes information that the specification hides from the user, but that the tester needs to know. For example, a user needn't know that a routine uses a hash table, but a tester would want to probe hash table collision handling. There are two steps:

Broken 1: The code is scanned, looking for important operations and types that aren't visible in the specification. Types are often recognized because of comments about the use of a variable. (A variable's declaration does not contain all the information needed; an integer may be a count, a range, a percentage, or an index, each of which produces different test conditions.) Cliched operations are often distinct blocks of code separated by blank lines or comments. Of course, the key way you recognize a cliche is by having seen it often before. Once found, these cliches are then treated exactly as if they had been found in the specification. No attempt is made to find and satisfy coverage conditions. (The name indicates this: the box is broken open enough for us to see gross features, but we don't look at detail.)

Broken 2: These new test conditions are combined into new test cases.

In production use, a tester presented with a specification and a finished program would omit step Black3. Test conditions would be derived from both the specification and the code, then combined together. This would minimize the size of the test suite. For this experiment, the two test suites were kept separate, in order to see what the contribution of looking at the code would be. This also simulates the more desirable case where the tests are designed before the code is written. (Doing so reduces elapsed time, since tests can be written while the code is being written. Further, the act of writing concrete tests often discovers errors in the specification, and it's best to discover those early.)

In this experiment, the separation is artificial. I wrote the specifications for six of the programs, laid them aside for a month, then wrote the test cases, hoping that willful forgetfulness would make the black box testing less informed by the implementation.

2.2.2. Applying Coverage

After the black and broken box tests were run, three coverage test suites were written. At each stage, cumulative coverage from the previous stages was measured, and a test suite that reached 100% feasible coverage on the next coverage goal was written and run. The first suite reached 100% feasible branch and loop coverage, the next 100% feasible multi-condition coverage, and the last 100% feasible weak mutation coverage. The stages are in order of increasing difficulty. There is no point in considering weak mutation coverage while there are still unexecuted branches, since eliminating the branches will eliminate many weak mutation conditions without the tester having to think about them.

In each stage, each coverage condition was first classified. This meant:

(1) For impossible conditions, an argument was made that the condition was indeed impossible. This argument was not written down (which reduces the time required).

(2) Only weak mutation conditions could be considered not worthwhile. The rule (necessarily imprecise) is given in the next section. In addition to the argument for infeasibility, an argument was made that the code was correct as written. (This was always trivial.) The arguments were not written down.

---

[2] The name was coined by Johnny Zweig. This stage is similar to Howden's functional testing: see [Howden80a], [Howden80b], or [Howden87].

(3) For feasible conditions, a test condition was written down. It described the coverage condition in terms of the unit's input variables.

After all test conditions were collected, they were combined into test cases in the usual way.

2.2.2.1. Feasibility Rules

Weak mutation coverage conditions were never ruled out because of the difficulty of eliminating them, but only when a convincing argument could be made that the required tests would have very little chance of revealing faults. That is, the argument is that the coupling effect will not hold for a condition. Here are three typical cases:

(1) Suppose *array* is an input array and array[0]=0 is the first statement in the program. If array[0] is initially always 0, GCT will complain that the initial and final value of the array are never different. However, a different initial value could never have any effect on the program.

(2) In one program, fopen() always returned the same file pointer. Since the program doesn't manipulate the file pointer, except to pass it to fread() and fclose(), a different file pointer would not detect a fault.

(3) A constant 0 is in a line of code executed by only one test case. In that test case, an earlier loop leaves an index variable with the value 0. GCT complains that the constant might be replaced by the variable. That index variable is completely unused after the loop, it has been left with other values in other tests, and the results of the loop do not affect the execution of the statement in question. Writing a new test that also executes the statement, but with a different value of the index variable, is probably useless.

2.2.2.2. Weak mutation coverage

Weak mutation coverage tools can vary considerably in what they measure. GCT began with the single-token transformations described in [Offutt88] and [Appelbe??], eliminating those that are not applicable to C. New transformations were added to handle C operators, structures, and unions. Space does not permit a full enumeration, but the extensions are straightforward. See [Agrawal89] for another way of applying mutation to C.

Three extensions increased the cost of weak mutation coverage:

(1) In an expression like (*variable* < *expression*), GCT requires more than that *variable*≠*alternate*. It requires that *variable*<*expression* ≠ *alternate*<*expression*. This *weak sufficiency requirement* guards against some cases where weak mutation would fail to find a fault; see [Marick90].

(2) Weak sufficiency also applies to compound operands. For example, when considering the operator **ptr*, GCT requires **ptr*≠**other_ptr*, not just *ptr*≠*other_ptr*. (Note: [Howden82] handles compound operands this way. It's mentioned here because it's not an obvious extension from the transformations given in [Offutt88], especially since it complicates the implementation somewhat.)

(3) Variable operands are required to actually vary; they cannot remain constant.

See [Agrawal89] for another way of applying mutation to C.

2.2.3. What was Measured

For each stage, the following was measured:

(1) The time spent designing tests. This included time spent finding test conditions, ruling out infeasible coverage conditions, deciding that code not worth covering (e.g., potential weak mutation faults) was correct, and designing test cases from test conditions. The time spent actually writing the tests was not measured, since it is dominated by extraneous factors. (Can the program be tested from the command line, or does support code have to be written? Are the inputs easy to provide, like integers, or do they have to be built, like linked lists?)

(2) The number of test conditions and test cases written down.

(3) The percent of coverage, of all types, achieved. This is the percent of total coverage conditions, not just feasible ones.

(4) The number of feasible, impossible, and not worthwhile coverage conditions.

2.2.4. An Example

LC is a 272-line C program that counts lines of code and comments in C programs. It contains 923 coverage conditions.

The manpage was used as the starting specification; 101 test conditions were generated from it. These test conditions could be satisfied by 36 test cases. Deriving the test conditions and designing the test cases took 2.25 hours. Four faults were found.[3]

These coverages were achieved in black box testing:

	Branch	Loop	Multi	Operator	Operand
Number satisfied	94 of 98	19 of 24	41 of 42	170 of 180	470 of 580
Percent	96%	79%	98%	94%	81%

The next stage was broken box testing. In two hours, 13 more test conditions and 4 more test cases were created. The increase is not large because the implementation of this program is relatively straightforward, with few hidden operations like hash table collision handling. No more faults were found, and the following increases in coverage were seen:

	Branch	Loop	Multi	Operator	Operand
Number newly satisfied	2 of 4	0 of 5	1 of 1	2 of 10	7 of 110
Cumulative Percent	98%	79%	100%	96%	82%

In the next stage, the seven unsatisfied branch and loop conditions required only 15 minutes to examine. Four were impossible to satisfy, two more were impossible to satisfy because of an already-found fault, and one could be satisfied. The resulting test satisfied exactly and only its coverage condition.

Because multi-condition coverage was 100% satisfied, 8 operator test conditions and 103 operand test conditions remained to be satisfied. Of these, 107 were infeasible. 97 of these were impossible (one of them because of a previously-discovered fault), and the remaining 10 were judged not worth satisfying.

Seven new test conditions were written down. Two of these were expected to satisfy the remaining four weak mutation conditions, and the rest were serendipitous. (That is, while examining the code surrounding an unsatisfied condition, I discovered an under-tested aspect of the specification and added tests for it, even though those tests were not necessary for coverage. This is not uncommon; often these tests probe whether special-case code needs to be added to the program. Note that [Glass81] reports that such omitted code is the most important class of fault in fielded systems.)

These seven test conditions led to four test cases. One of the serendipitous test conditions discovered a fault.

Satisfying weak mutation coverage required 3.25 hours, the vast majority of it devoted to ruling out impossible cases.

[3] The program has been in use for some years without detecting these faults. All of them corresponded to error cases, either mistakes in invocation or mishandling of syntactically incorrect C programs.

3. Results

This section presents the uninterpreted data. Interpretations and conclusions are given in the next section.

Measures of effort are given in this table. All measures are mean cumulative percentages; thus weak mutation always measures 100%.

	Black	Broken	Branch+Loop	Multi	Weak
Time	53	74	76	77	100
Test Conditions	79	93	95	95	100
Test Cases	74	89	92	92	100

The next table reports on coverage achieved. Numbers give the percent of total coverage conditions eliminated by testing. An asterisk indicates that all coverage conditions of that type were eliminated (either by testing or because they were infeasible). One number, the 100% for All Path-Based Coverage in the Branch+Loop stage, has a different interpretation. It measures the branches and loops eliminated either by testing or inspection, together with the multi-conditions eliminated by testing alone. This was done because the number should indicate how much remains to be done after the stage.

	Black	Broken	Branch+Loop	Multi	Weak
Branch Coverage	95	99	*	*	*
All Path-based Coverage	92	95	100	*	*
Weak Coverage	84	88	89	89	*

The time spent during the latter stages depends strongly on how many coverage conditions have to be examined. Most were weak mutation conditions: 93% (std. dev. 3%), compared to 5% (std. dev. 3%) loop, 1% (std. dev. 2%) branch, and 0.1% (std. dev. 0.3%) multi-condition.

Because examining an impossible condition is of little use, it is useful to know what proportion of time is spent doing that. This was not measured, but it can be approximated by the proportion of examined conditions which were impossible.

	Branch	Loop	Multi	Weak
Percent Impossible	83	70	0	69
Percent Not Worth Testing	0	0	0	17
Feasible	17	30	100	14

The infeasible weak mutation conditions were the vast majority of the total infeasible conditions (mean 94%, std. dev 5). Of these, 9% (std. dev. 8) were operator conditions, 44% (std. dev. 25) were due solely to the requirement that variables vary, and other operand conditions were 47% (std. dev. 19). The actual significance of the "variables vary" condition is less than the percentage suggests; ruling them out was usually extremely easy.

In any process, consistency of performance is important. This table shows the standard deviations for the first two stages. (The numbers in parentheses are the mean values, repeated for convenience.) For example, during black box testing, 79% of the test conditions were written, with an 18% standard deviation. After broken box testing, the percentage increased to 93% and the standard deviation decreased to 5%. Results for later stages are not given because the mean values are very close to 100% and the variability naturally drops as percentages approach 100%. For the same reason, the apparent decreases shown in this table may not be real. Because the stages are not independent, there seems to be no statistical test that can be applied.

	Black	Broken
Time	19 (53)	14 (74)
Test Conditions	18 (79)	5 (93)
Test Cases	11 (74)	8 (89)
Branch Coverage	6 (95)	3 (99)
All Path-based Coverage	7 (92)	2 (95)
Weak Coverage	11 (84)	4 (88)

The mean absolute time in minutes per line of code is given in this table. The values are cumulative from stage to stage.

	Black	Broken	Branch+Loop	Multi	Weak
Minutes/LOC	2.4	2.8	2.9	2.9	3.6
Std. Deviation	1.4	1.4	1.4	1.4	1.4

4. Discussion

This section first interprets the results, then draws some conclusions about the use of coverage in testing.

Measured by branch coverage alone, the first two stages attained high coverage: 95% for black, 99% for broken[4]. These numbers are higher than those reported in other studies of unit-sized programs. [Vouk86] achieved 88% branch coverage with black-box testing of several implementations of a single specification. [Lauterbach89] found 81% coverage for units selected from production software.

When all of the path-based coverage measures are considered together, the results are similar: 92% for black, 95% for broken. The additional cost to reach 100% on all of the path-based coverage measures was 3% of total time, 2% of total test conditions, and 3% of the total test cases. (If testing had not included weak mutation coverage, the percentages would have been 4% of time, 3% of test conditions, and 4% of test cases.) Coverage testing raised the design cost from 2.8 to 2.9 minutes per line of code. Given that test writing time is proportional to test design time, and that the fixed startup cost of testing is large, this extra expense is insignificant. This result is similar to that of [Weyuker90], who found that stricter dataflow criteria were not much more difficult to satisfy than weaker ones, despite much larger theoretical worst-case bounds.

High path-based coverage is a reasonable expectation. The cost to reach 100% feasible coverage is so low that it seems unwise to let a customer be the first person to exercise a branch direction, a particular loop traversal, or a multi-condition.

The black and broken stages lead to a lower weak mutation coverage: 84% for black and 88% for broken. [DeMillo88] cites an earlier study where black box tests yielded 81% mutation coverage for a single program.

Continuing to branch and loop coverage gained 1%, leaving 11% of the weak mutation conditions. This agrees with [Ntafos84], who found that branch coverage (achieved via random testing) left on average 8% uncovered mutation conditions. (A variant of dataflow testing left 4% uncovered conditions.) It does not agree so well with the study cited in [DeMillo88]. There, branch coverage of a simple triangle classification program left 22% uncovered mutation conditions.[5]

After the multi-condition stage, satisfying the remaining weak mutation conditions is expensive: 8% more test cases and 5% more test conditions, but at the cost of 23% of the total time. The large time is because most coverage conditions (93%) are weak mutation. The relatively low yield is because most of the remaining weak mutation conditions are impossible (69%) or not worth testing (17%). The time spent ruling these out is wasted. GCT could be enhanced to do more of this automatically, but considerable manual work is unavoidable. Further, weak mutation coverage conditions are the hardest to eliminate; the effort is typically greater than, say, forcing a branch to be taken in the TRUE direction. Thus, the cost of weak mutation testing is likely to remain higher.

Of course, those remaining expensive tests might be quite cost effective. They might find many faults. In the near future, GCT and this testing technique will be applied to programs during development. These case studies will determine whether the extra effort of weak mutation coverage is worthwhile by measuring how many faults are detected. In the meantime, weak mutation coverage cannot be recommended.

There is one exception. Relational operator faults (< for <= and the like) are common; indeed, they are the motivation behind testing boundary conditions. As [Myers78] observes, testers often think they are doing a

[4] Recall again that I wrote the specifications. The black box numbers are less reliable than the broken box numbers. Recall also that the percentages reported are of total conditions, not just the feasible ones.

[5] Note that these two studies report on strong mutation coverage. However, empirical studies like [Marick90] and [Offutt91] have found that strong mutation coverage is roughly equal to weak mutation coverage.

better job testing boundary conditions than they actually are. The relational operator subset of operator weak mutation coverage provides an objective check. Although no records were kept for this subset, few of these conditions remained until the weak mutation stage, and they were generally easy to satisfy. GCT is in the early stages of production use within Motorola, and users are advised to follow the technique described here through branch, loop, multi-condition, and relational operator coverage.

This advice may seem peculiar in one respect. General weak mutation testing is ruled out because it does not seem useful enough for the effort expended. However, broken box testing gains less than 5% coverage but costs 21% of the time, along with 15% of the test cases. It might seem that broken box testing is not worthwhile, but recall how the variability in test conditions and coverage conditions drops sharply from black to broken box testing. This may be an artifact of using percentages. However, it is certainly plausible -- broken box testing removes one source of variability, the proportion of internal cliches exposed in the specification. Further, most of the test conditions in broken box testing exist to discover faults of omission: to uncover missing code, rather than to exercise present code. One must remember that coverage, while an important estimator of test suite quality, does not tell the whole story. Like all metrics, it must be used with care, lest the unmeasured aspects of quality be forgotten. The effective use of coverage will be discussed later, after some other important issues are considered.

4.1. Potential Problems with this Study

The coverage in this study is generally higher than that found in other studies. To what extent are these results particular to this technique? Three factors might be important:

(1) The rigorous, stereotyped format of the specification makes it less likely that details needing testing will be lost in the clutter. The same is true of the methodical procedure for deriving test cases. Other formats and procedures should work as well.

(2) The catalog used in black and broken box testing is a collection of the test conditions an expert tester might use, for the benefit of novices and experts with bad memories. It probably contributes to high coverage, but the focus of the catalog is faults of omission -- and tests to detect omitted code have no effect on coverage.

(3) Broken box testing brings tester-relevant detail into the specification. It has an effect on coverage (a 3-4% increase in the different coverage measures).

In short, this technique is a codification of existing practice, designed to make that practice more consistent and easier to learn. Other methodical codifications should achieve comparable coverages.

In isolation, this study is too small for firm conclusions. Not enough programs were tested, and they were all tested by one person, who had written the specifications. However, the results are consistent with other experience. In a later experiment, a classful of students applied an extension of this technique to three programs. All the data has not been analysed, but the same pattern appears. The next study will apply the technique to a complete subsystem. It will be used to refine the technique, to try to repeat these results, and to address two additional questions: how well do the different stages detect faults? and what is the effect of differing definitions of feasibility?

4.2. The Effective Use of Coverage

100% feasible coverage appears to be a reasonable goal. How should it be achieved? When coverage is first measured, there will be uncovered conditions. How are they to be handled?

The natural impulse is to add tests specifically designed to increase coverage. This approach, however, is based on a logical fallacy. Because we believe that (1) a good test suite will achieve high coverage, we are also asked to believe that (2) any test suite that achieves high coverage must be good. Yet (1) does not imply (2). In particular, tests designed to satisfy coverage tend to miss faults of missing code, since a tool cannot create coverage conditions for code that ought to be there, but isn't. These are the very faults that [Glass81] found most important in fielded systems.

An unexercised coverage condition should be considered a signal pointing to an under-exercised part of the specification. New test conditions should be derived from that part, not just from the coverage condition that points to it. This may seem an unnecessarily thorough approach, but the results of this study suggest that its cost is only a few percent of the total cost of test design. And, as we saw in the LC example, such "unnecessary" test cases may be the ones that find faults, while the test cases added for coverage do not.

A more important question is this: what is to be done if coverage is substantially lower than 100%?

If high coverage is not achieved, that's not just a signal to improve the test suite, it's also a signal to improve the test *process*. Suppose the process leads to 60% branch coverage. The remaining coverage conditions can still be used to produce a good test suite, so long as they are treated as pointers into the specification. But it would have been better to produce this good test suite in the first place:

(1) Tests added later are more expensive. Time must be spent understanding why a coverage condition is unexercised.

(2) Tests sometimes discover problems with the specification. It is better to discover those problems before the code is written.

(3) If coverage varies widely from program to program, the cost and duration of testing is less predictable.

Low coverage should lead to a diagnosis of a process problem. Perhaps the test generation technique needs to be improved, or the tester needs better training, or special difficulties in testing this application area need to be addressed. A common problem is that the form or style of the specification obscures or ignores special cases.

4.3. Testing Large Systems

The results of this paper apply only when test cases are derived from individual units. If this same technique is applied to collections of units, or whole systems, the result will be lower coverage. A subsystem or system specification will not contain enough detail to write high-yield test cases.

However, testing each function in isolation is very expensive. In the worst case, special support code ("test harnesses") will have to be written for every function. The cost of support code can dominate the cost of unit testing.

A good compromise is to use subsystems as test harnesses. All routines in a subsystem are tested together, using test conditions derived from their specifications, augmented by test conditions targeted to integration faults. Tests are then added, based on coverage data.

Disadvantages of this approach are:

(1) Extra design effort to discover how to force the subsystem to deliver particular values to the function under test.

(2) Faults found by the tests will be more difficult to isolate.

(3) More coverage conditions will be impossible. For example, the subsystem might not allow the exercising of a function's error cases. Faults in error handling might not be discovered until this "tested" function is reused.

(4) Increased chance that a possible condition will be thought impossible.

Offsetting these is the advantage of not having to write harnesses for all of the functions. The use of the subsystem as the single harness also makes the creation and control of a repeatable test suite easier. This approach should usually be reasonable with subsystems of up to a few thousand lines of code. Raw size is not as important as these factors:

(1) Clean boundaries between the subsystem and other subsystems. This reduces the cost of building a harness for the subsystem.

(2) Some support for testing built into the subsystem. Relatively simple changes can often make testing much easier. Because the changes are simple, they can often be retrofitted.

(3) A single developer responsible for the subsystem. This reduces the cost of test design and sharply reduces the cost of diagnosis.

Once coverage has been achieved at the subsystem level, it need not be repeated at large-scale integration testing or system testing. That testing should be targeted at particular risks, preferably identified during system analysis and design. 100% coverage is not relevant.

This strategy, "unit-heavy", is probably not often done. Usually more effort is devoted to integration and (especially) system testing. Sometimes unit testing is omitted entirely. Other times it is done without

building test harnesses or repeatable test suites, which essentially means omitting unit testing during maintenance. The reason for this "unit-light" strategy is a risk/benefit tradeoff: the extra cost of discovering faults later is expected to be less than the cost of more thorough early testing.

What relevance has this paper to that strategy? The results do not apply. However, it may provide more evidence that the unit-heavy strategy is reasonable:

(1) It provides a measurable stopping criterion, 100% feasible coverage, for testing. Testers often suffer from not having concrete goals.

(2) The criterion is intuitively reasonable. It makes sense to exercise the branches, loops, and multi-conditions, and to rule out off-by-one errors.

(3) As an objective measure, the criterion can be used to achieve more consistent and predictable testing.

(4) The cost can be reduced by using subsystem harnesses, and can be greatly reduced if testing is considered during design.

However, there is no hard data on which to base a choice. Further studies are being planned to get a clearer idea of the relative costs and benefits of the two strategies (which, of course, form a continuum rather than two distinct points). In the meantime, programmers, testers, and managers can examine bug reports from the field and ask

(1) Would 100% coverage have forced the early detection of this fault?

(2) Would thorough black box tests have forced the detection of the fault? (A somewhat more subjective measure.)

[Howden80a] took this approach with 98 faults discovered after release of edition five of the IMSL library. He found that black box testing would have found 20% of the faults, applying black box techniques to internals would have found 18%, and branch coverage would have forced 13%. (Some faults would be found by more than one technique.) There is danger in extrapolating these numbers to other systems: they are for library routines, they depend on how the product was tested before release, and they do not describe the severity of the faults. There is no substitute for evaluating your own testing process on your own products.

Appendix A: An example specification

This is a part of the specification for the compare_files() routine in GNU diff. The actual specification explains the context.

```
COMPARE_FILES(DIR1, FILE1, DIR2, FILE2, DEPTH)
        All arguments are strings, except DEPTH, which is an integer.

PRECONDITIONS:
1.      Assumed: At most one of FILE1 and FILE2 is null.

2.      Assumed: If neither of FILE1 and FILE2 is null
                 THEN they are string-equal.

[...]
4.      Validated:  if FILE1 is non-NULL, then file DIR1/FILE1 can be
                    opened for reading
        On failure:
                    An error message is printed to standard error.
                    The return value is 2.

[...]

POSTCONDITIONS:
1       IF FILE1 and FILE2 are plain files that differ
        THEN the output is a normal diff:
```

```
1c1
< bar
---
> foo
```

and compare_files returns 1.

2. If FILE1 and FILE2 are identical plain files
   THEN there is no output and the return value is 0.

3. IF FILE1 is a directory, but FILE2 is not
   THEN
   the output is "FILE1 is a directory but FILE2 is not"
   the return value is 1.

[...]

Appendix B: Example of catalog entries

This appendix shows the catalog entries that apply to the examples given in the text.

23. GENERAL SEARCHING CONDITIONS

- Match not found
- Match found (exactly one match in collection)
- More than one match in the collection
- Single match found in first position DEFER
  (it's not the only element)
- Single match found in last position DEFER
  (it's not the only element)

Note that these conditions apply whether the search is forward or backward.

There is more detail to the technique than explained in this paper, aimed toward reducing the amount of work; the DEFER keyword is part of that detail.

19. PATHNAMES

...

19.1. Decomposing Pathnames

There are many opportunities for errors when decomposing pathnames into their component parts and putting them back together again (for example, to add a new directory component, or to expand wildcards).

- <text>/
- <text>/<text>
- <text>/<text>/<text>
- <text>//<text>

Also consider the directory and file components as Containers of variable-sized contents.

REFERENCES

[Agrawal89]
H. Agrawal, R. DeMillo, R. Hathaway, Wm. Hsu, Wynne Hsu, E. Krauser, R.J. Martin, A. Mathur, and E. Spafford. *Design of Mutant Operators for the C Programming Language*. Software Engineering Research Center report SERC-TR-41-P, Purdue University, 1989.

[Appelbe??]
W.F. Appelbe, R.A. DeMillo, D.S. Guindi, K.N. King, and W.M. McCracken, *Using Mutation Analysis For Testing Ada Programs*. Software Engineering Research Center report SERC-TR-9-P, Purdue University.

[Beizer83]
Boris Beizer. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1983.

[DeMillo78]
R.A. Demillo, R.J. Lipton, and F.G. Sayward, ''Hints on test data selection: help for the practicing programmer''. *Computer*. vol. 11, no. 4, pp. 34-41, April, 1978.

[DeMillo88]
R.A. DeMillo and A.J. Offutt. ''Experimental Results of Automatically Generated Adequate Test Sets''. *Proceedings of the 6th Annual Pacific Northwest Software Quality Conference*, Portland, Oregon, September 1988.

[DeVor91]
R.E. DeVor, T.H. Chang, and J.W. Sutherland. *Statistical Methods for Quality Design and Control*. New York: MacMillan, 1991 (in press).

[Glass81]
Robert L. Glass. ''Persistent Software Errors''. *Transactions on Software Engineering*, vol. SE-7, No. 2, pp. 162-168, March, 1981.

[Hamlet77]
R.G. Hamlet. ''Testing Programs with the aid of a compiler''. *IEEE Transactions on Software Engineering*, vol. SE-3, No. 4, pp. 279-289, 1977.

[Howden78]
W. E. Howden. ''An Evaluation of the Effectiveness of Symbolic Testing''. *Software - Practice and Experience*, vol. 8, no. 4, pp. 381-398, July-August, 1978.

[Howden80a]
William Howden. ''Applicability of Software Validation Techniques to Scientific Programs''. *Transactions on Programming Languages and Systems*, vol. 2, No. 3, pp. 307-320, July, 1980.

[Howden80b]
W. E. Howden. ''Functional Program Testing''. *IEEE Transactions on Software Engineering*, vol. SE-6, No. 2, pp. 162-169, March, 1980.

[Howden82]
W. E. Howden. ''Weak Mutation Testing and Completeness of Test Sets''. *IEEE Transactions on Software Engineering*, vol. SE-8, No. 4, pp. 371-379, July, 1982.

[Howden87]
W.E. Howden. *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

[Johnson83]
W.L Johnson, E. Soloway, B. Cutler, and S.W. Draper. *Bug Catalogue: I.* Yale University Technical Report, October, 1983.

[Lauterbach89]
L. Lauterbach and W. Randall. ‘‘Experimental Evaluation of Six Test Techniques’’. *Proceedings of COMPASS 89*, Washington, DC, June 1988, pp. 36-41.

[Marick90]
B. Marick. *Two Experiments in Software Testing*. Technical Report UIUCDCS-R-90-1644, University of Illinois, 1990. Portions are also to appear in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*.

[Myers78]
Glenford J. Myers. ‘‘A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections’’. *Communications of the ACM*, Vol. 21, No. 9, pp. 760-768, September, 1978.

[Myers79]
Glenford J. Myers. *The Art of Software Testing*. New York: John Wiley and Sons, 1979.

[Ntafos84]
Simeon Ntafos. ‘‘An Evaluation of Required Element Testing Strategies’’. *Proceedings of the 7th International Conference on Software Engineering*, pp. 250-256, IEEE Press, 1984.

[Offutt88]
A.J. Offutt. *Automatic Test Data Generation*. Ph.D. dissertation, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

[Offutt89]
A.J. Offutt. ‘‘The Coupling Effect: Fact or Fiction’’. *Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification*, in *Software Engineering Notes*, Vol. 14, No. 8, December, 1989.

[Offutt91]
A.J. Offutt and S.D. Lee, *How Strong is Weak Mutation?*, Technical Report 91-105, Clemson University Department of Computer Science, 1991. Also to appear in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*.

[Perry89]
Dewayne E. Perry. ‘‘The Inscape Environment’’. *Proceedings of the 11th International Conference on Software Engineering*, pp. 2-12, IEEE Press, 1989.

[Rapps85]
Sandra Rapps and Elaine J. Weyuker. ‘‘Selecting Software Test Data Using Data Flow Information’’. *Transactions on Software Engineering*, vol. SE-11, No. 4, pp. 367-375, April, 1985.

[Rich90]
C. Rich and R. Waters. *The Programmer’s Apprentice*. New York: ACM Press, 1990.

[Spohrer85]
J.C. Spohrer, E. Pope, M. Lipman, W. Scak, S. Freiman, D. Littman, L. Johnson, E. Soloway. *Bug Catalogue: II, III, IV*. Yale University Technical Report YALEU/CSD/RR#386, May 1985.

[Su91]
Jason Su and Paul R. Ritter. ‘‘Experience in Testing the Motif Interface’’. *IEEE Software*, March, 1991.

[Vouk86]
Mladen A. Vouk, David F. McAllister, and K.C. Tai. ‘‘An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software’’. In *Proceedings of the Workshop on*

Software Testing Conference, pp. 74-81, Banff, Canada, 1986.

[Weyuker90]
Elaine J. Weyuker. ‘‘The Cost of Data Flow Testing: An Empirical Study’’. *Transactions on Software Engineering*, vol. SE-16, No. 2, pp. 121-128, February, 1990.

[Zang91]
Xiaolin Zang and Dean Thompson. ‘‘Public-Domain Tool Makes Testing More Meaningful’’ (review). *IEEE Software*, July, 1991.