THE WEAK MUTATION HYPOTHESIS

Brian Marick
Motorola, Inc.

Abstract

In fault–adequate testing, a fault is detected if a test case satisfies three conditions. If the reachability condition is satisfied, the fault is executed. If the necessity condition is satisfied, the fault has an effect on the program state. If the sufficiency condition is satisfied, the effect can be observed in the output. The *weak mutation hypothesis* is that test cases that satisfy the reachability and necessity conditions will satisfy the sufficiency condition. A study of 100 faults reveals that the hypothesis holds true for 44 of them. 16 faults would be detected except under exceptional circumstances, such as a random block of memory containing a particular string, so the hypothesis is said to "almost hold" for 60 faults. If a very weak sufficiency condition with local scope is added, the hypothesis almost holds for 70.

Weak mutation coverage requires that the reachability and necessity conditions be satisfied for all potential fault locations in the program, not just for the actual fault. Test cases satisfying the conditions for other locations might force the detection of the fault. A simulation using branch coverage suggests that 16 of the remaining 30 faults would be detected in this way. If the test cases were developed using standard black–box testing techniques, they would detect 6 of the remaining 14 faults.

In sum, given 100 faults that fault–adequate testing would be certain to detect, a combination of black box testing and weak mutation coverage could reasonably be expected to detect 92 of them.

Brian Marick is the recipient of a Motorola Partnerships in Research Grant. 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. Background

A *fault* is a change in text that causes a correct program to be incorrect. For example, if the statement

    if (A < B)

should be

    if (A <= B)

the fault is the replacement of <= with <. The goal of fault–adequate testing [Morell90] is to guarantee the absence of particular kinds of faults in a program. For example, given a program, one wishes to generate a test suite that would detect if any relational operator should have been replaced by any other.

In *strong mutation testing* [Demillo78] [SERC87], potential faults (or corrections to faults) are injected into the program to form variant programs. The test suite is augmented until every (*program*, *variant*) pair produces a different result on at least one test case. (The results must still be compared against the specification to determine whether it is the original program or the variant that is in error.) Such a test case must satisfy three conditions:

the reachability condition
: This condition causes the location of the potential fault to be executed. (In the above example, the comparison operation must be invoked.)

the necessity condition
: This condition causes the effect of the fault to be detectable at the point of the fault. (The comparison operator must produce different results in the original and variant programs.)

the sufficiency condition
: This condition requires that the local effect

remain detectable at program exit.

This terminology is due to [Offutt88]. See also [Morell83], who uses different terms.

A difficulty with mutation testing is that it requires the creation and execution of a large number of variant programs. Consequently, *weak mutation testing* has been proposed [Howden82]. Its requirements are less strict: for each potential fault, there must be a test case that satisfies the reachability and necessity conditions. The advantage is that weak mutation can be determined by instrumenting a single program, rather than by executing a large number of them. In the example, the conditional test could be replaced by

if (INSTRUMENT_LESS(A, B, (A < B)))

where *INSTRUMENT_LESS* is a function that records whether its two arguments are equal (that is, records whether the necessity and reachability conditions for the potential fault are satisfied) and returns the value of the original expression.

The disadvantage of weak mutation testing is that there is no guarantee that the different immediate effect will cause a different final result. *Firm mutation testing* [Woodward88] is an intermediate technique. Instrumentation is not at the point of the fault, but at some later point or points in the program. If the difference is visible there, it is more likely to remain visible at program exit.

Weak mutation is as effective as strong mutation if the *weak mutation hypothesis* is true:

(*reachability* and *necessity*) implies *sufficiency*

In his dissertation, Offutt conducted an experiment that suggests the weak mutation hypothesis is true more than 61% of the time [Offutt88]. However, this experiment was on programs smaller than those used in industry, and it did not consider a certain class of faults (those whose reachability and necessity conditions involved internal variables). Therefore, another experiment seemed worthwhile.

1.1. The Effect of the Rest of the Program

Suppose that the weak mutation hypothesis does not hold for a particular fault. That is, there exists a non-empty input set that satisfies the reachability and necessity conditions while not producing a detectable failure. But in the code under test, there will be many locations with potential faults, each with its own reachability and necessity conditions. It may be that satisfying those other conditions will force the execution of the original fault in a way that must produce a detectable failure.

Thus, the weak mutation hypothesis may not hold when a fault is considered alone, but may hold when the fault is considered as part of a larger program.

More formally, call the reachability condition for the fault in question *fault−reachability* and the reachability condition for some other location *other−reachability*. *fault−necessity* and *other−necessity* denote the necessity conditions. For convenience, *other−weak* is used as shorthand for (*other−reachability* and *other−necessity*).

We consider only locations for which *other−reachability implies fault−reachability*; that is, reaching the other location requires executing the known fault. If this is not true for a particular location, it cannot force the detection of the fault.

There are two cases:

CASE 1: If *other-weak* implies (*fault-necessity* and *fault-sufficiency*) ruling out a potential fault at the other location requires a test case that causes a failure.

CASE 2: The fault may also be detected because it prevents weak mutation coverage at another location. This might happen in two ways:

Condition	**Failure to satisfy means**
other−reachability	the fault prevents the other location from ever being reached.
other−weak	the fault prevents the other location's necessity condition from ever being measured true.

When a condition is not satisfied, the program is examined to determine how to satisfy it. The tester realizes that it cannot be satisfied and identifies the fault as the reason. (There is, of course, a chance of human error: some conditions in correct programs are unsatisfiable, and an unsatisfiability due to a fault might be attributed to that.)

1.2. The Rest of the Paper

The next section describes results from the literature. The following three sections describe the experiment, the data, and conclusions. This study considers a particular type of fault; the final section extends the ideas of weak mutation to other types.

2. Related Studies

There have been other studies of test effectiveness. They are of two types: studies, like this one, of injected faults, and studies of naturally occurring faults. Various results are presented in Table 1; branch testing and a combined technique are shown for comparison. Notes:

(1) The results in [Girgis86] are not directly applicable to the weak mutation hypothesis. In that study, a test was written. If the fault was detected, all techniques whose criteria that test satisfied were given credit. If not, another test was written. Thus, the results should be interpreted as "59% of the time, the fault was not **detected** by weak mutation testing," not "59% of the time, the fault was not **detectable** by weak mutation testing."

(2) The combined method used in [Howden78] was specification-based testing, three static analysis techniques, and structured testing, which forces branch coverage plus the repeated execution of every loop.

(3) In the [Lauterbach89] case, only the results for unit testing are reported, as they are most comparable to the other studies.

The studies are inconclusive. Full mutation testing seems to have a high effectiveness, as does a combined technique. The effectiveness of branch testing is quite varied.

3. The Experiment

Five sizable and widely-used programs were chosen. Four were used because they are readily available, allowing others to replicate these results. The fifth is well-known to be complex and difficult to test. The programs are shown in Table 2. The size is in lines, including comments and blank lines.

Four routines were chosen at random from each program.[2] The routines were between 9 and 206 lines of code[3], with a mean of 51 and a standard deviation of 58. Each routine was used in five trials.

[1] UNIX is a trademark of AT&T.

[2] Routines with five or fewer non-comment source lines were discarded. Two routines, one with 230 lines and one with 185 lines, were discarded because they were too complex and dependent on the rest of the program to analyse accurately by hand.

[3] Excluding blank lines, comments, and lines containing only braces. The routine header counts as 1 line.

Study	Seeded?	Faults	Branch	Weak Mutation	Full Mutation	Combined
[Girgis86]	Yes	80	34%	41%	–	–
[Ntafos84]	Yes	468	91.6%	–	–	–
[Howden78]	1 of 28	28	21%	–	–	89%
[Howden80]	No	98	13%	–	–	–
[Lauterbach89]	No	85	25%	–	–	–
[Budd80]	No	25	–	–	80%	–
[DeMillo88]	Yes	962	77.8%	–	99.7%	–
[Hennell84]	No	9	77.7%	–	–	–

Table 1: Test Effectiveness Studies

Name	Version	Size	Description
gcc	1.36	96000	C compiler
GNU make	3.57	25000	dependency tracking language
PERL	3.0	26000	text–processing language
RCS	4.3	12000	version control system
UNIX[1] System V	3.2	101000	operating system kernel

Table 2: Programs Used

In each trial, a random fault of the type used in Offutt's thesis was injected (by hand) into the routine. Notice that these faults are those simple faults guaranteed to be detected by strong mutation. This is appropriate, since the purpose of this study was to compare weak to strong mutation. However, such faults appear to be a minority of faults found in programs [Marick90].

If the fault produced an equivalent program, one which computes the same results for all inputs, it was discarded and a new fault was inserted. This happened in three cases.

The routines were then inspected to determine under what circumstances the effect of the fault would be observable. (Although it is well known that observable differences are often not observed [Basili87] [Myers78], that problem is independent of the testing technique.) An observable effect is any difference in the resulting state. This includes the return value, changes to global variables, or I/O. Differences in the addresses of returned data structures were not counted.

Faults were classified into the following categories. Each category includes only those cases that do not fit in a previous (stronger) category. Thus, COVERAGE HOLDS includes no cases where the weak mutation hypothesis holds, and SPEC LIKELY counts only cases where coverage does not hold.

HOLDS

The weak mutation hypothesis holds — the fault will be detected by any test case that satisfies the reachability and necessity conditions.

ALMOST HOLDS

The weak mutation hypothesis will hold except in an unlikely circumstance. Some circumstances are unlikely for any program: a typical example is a fault that will be detected unless an uninitialized pointer happens to point to a location in memory that contains data identical to the data pointed to in the correct program. This example is, in fact, the most common justification for ALMOST HOLDS.

But it is often the case that what is unlikely depends on the program's input and processing. For example, one fault would go unnoticed if tests used filenames that, in the correct program, always hash to the same location. Since the point of hashing is to make that unlikely, weak mutation was judged to ALMOST HOLD. In another case, the fault would be located unless unless a file named -q or -r existed, had the correct format, but caused no output.

Given the diversity of programs and their failures, judgements of likelihood are inevitably subjective. [Marick90] contains a description of each fault, together with justification for its classification.

COVERAGE HOLDS

If the fault is in neither of these two categories, the weak mutation hypothesis does not hold for it alone. However, satisfying the weak mutation conditions for other locations might reveal the fault. For this experiment, we consider all other locations in the routine containing the fault, but not those in other parts of the program. Measuring coverage in one routine at a time is typical of unit testing, when weak mutation is most likely to be used.

Checking weak mutation conditions by hand for all locations in a routine is impractical, but it can be approximated by determining whether branch coverage will force detection of the fault. Of course, branch coverage does not imply weak mutation coverage, so it serves as a lower bound.

As an example of this use of branch coverage, consider this C code. The comment describes how to create a faulty program.

```
A = 3 + B;     /* Replace B with B+1. */
if (C)
   A = 0;
output(A);
```

In this case, the necessity condition is always satisfied, since $B \neq B+1$. The weak mutation hypothesis does not hold for test cases where C is true. However, requiring that the branch be taken false will reveal the fault because an incorrect value of A will be written.

(Note: strictly, branch coverage does not provide a lower bound, since weak mutation coverage does not imply branch coverage. In the previous example, C's necessity condition requires that it be distinct from all other variables. This might be satisfied without ever requiring that C be false. However, such cases are probably rare in practice. Further, a tool which measures weak mutation coverage can easily be extended to measure branch coverage.)

A fault is also considered found if a branch cannot be taken in a particular direction. As an example, consider

```
A = 5;      /* Replace A with C */
if (C > 0)
   A = 0;
```

The program with the fault prevents the branch from ever being taken in the false direction.

In either of these two cases, we say that COVERAGE HOLDS. To avoid a proliferation of categories, we also say that COVERAGE HOLDS if branch coverage would force detection of the fault except in unlikely circumstances.

As noted earlier, some conditions in correct programs are unsatisfiable, and an unsatisfiability due to a fault might be attributed to that. This is much less likely for branch coverage than for weak mutation coverage, so it should not affect the lower bounds provided here.

SPEC LIKELY

In some cases where branch coverage does not force detection of the fault, it would be detected by ordinary specification-based testing techniques like those in [Myers79]. As is the case with ALMOST HOLDS, there is no hard-and-fast rule for deciding whether a fault belongs to this category. However, discovering these faults would be quite straightforward if tests were written following these rules:

- Both error and normal cases are tested.
- The program's input is partitioned into equivalence classes, and at least one test is written for each class. For example, if "white space" is described as blanks or tabs, tabs appear in some test case.
- Tests are written for off-by-one errors (boundary cases).

DOES NOT HOLD

Coverage is not adequate and a reasonable test suite might not detect the fault.

3.1. Weak Sufficiency

While executing the experiment, it was apparent that some (eventually, 10) failures to hold occurred with the same kind of fault, namely, the replacement of a variable or constant with another variable or constant. Consider, for example, this case:

```
if (var == constant)
  ..
else
  ..
```

A fault is the replacement of *constant* with *constant2*. The necessity condition in this case is trivial: $constant \neq constant2$. Table 3 shows the possible outcomes of the original program and the variant. Because both programs take the else branch in one case, the weak mutation hypothesis does not hold. However, if the necessity condition is augmented by a very weak sufficiency condition, that $(var{=}constant) \neq (var{=}constant2)$, that case is ruled out and weak mutation (or, more precisely, a version of firm mutation) holds.

A more rigorous statement of the *weak sufficiency condition* is:

(1) If the fault is that of replacing a data reference or constant by another data reference or constant,

(2) and the replaced unit is the entire left-hand or right-hand side of a relational expression,

	var=constant	var=constant2	var=other value
original	then branch taken	else branch taken	else branch taken
with fault	else branch taken	then branch taken	else branch taken

Table 3: Weak Sufficiency

Holds	Almost Holds	Coverage Holds	Spec Likely	Does Not Hold
50%	20%	16%	6%	8%

Table 4: Results of the experiment

(3) then the necessity condition for the unit is supplemented by the requirement that the relational expression take on a different value in the original and variant programs.[4]

This is a very reasonable condition, because it requires no additional instrumentation above that already required to detect wrong-relational-operator faults. The results reported here assume test cases satisfy weak sufficiency.

4. Results

Table 4 shows the results of the experiment.

A combination of weak mutation, branch testing, and black-box testing appears to be an effective testing method.

Various factors that might affect detectability were examined using a χ^2 test [Pearson04] with $p < 0.05$. For each of the following factors, we could not reject the hypothesis that a fault's detectability was independent of the factor:

(1) whether the fault was in a loop.

(2) whether the fault was in a routine with more lines than the mean.

(3) whether the fault was in a routine with cyclomatic complexity [McCabe76] less than 10.

(4) whether the fault was in a path selection expression.

(5) whether the fault was a data fault, an operator fault, or a miscellaneous fault.

(6) whether the fault seemed typical of those programmers make, one that a programmer would be unlikely to make, or one that a programmer would almost certainly not make.

One factor was significant. Coverage would have detected only 67% of the faults in routines whose only effect was to return a boolean value. It would have detected 86% of all faults. It may be that the smaller the number of distinct possible results, the greater the chance that a faulty program will serendipitously produce the correct result for a particular test case.

[Marick90] contains more details on the statistics and the experiment, including the text of the publicly-available programs, the faults inserted, and the classification of each fault.

5. Conclusions

Specification-based testing will surely be a part of any testing effort. Tests designed to satisfy coverage are relatively poor at detecting missing code [Basili87]. The reason is simple: a tool cannot create necessity conditions for code that ought to be there, but isn't. Note that [Glass81] found such faults to be most important in fielded systems.

This experiment suggests that the combination of specification-based and weak mutation testing will discover in excess of 90% of the faults that strong mutation testing would discover.

This conclusion assumes that weak mutation will be measured for an entire routine. Weak mutation testing, like mutation testing, is expensive. It might be desirable to apply it only to high-risk sections of routines, such as newly added code. In such a case, there would be some loss of

[4] Weak sufficiency is a weaker version of the *predicate constraints* from [Offutt88].

effectiveness, perhaps as much as 16%. However, the importance of that loss depends on the effectiveness of strong mutation itself, which is the topic of the next section.

6. Future Work: Extending Weak Mutation Testing

There are two potential problems with mutation testing:

the possibility of low effectiveness
: [Marick90] found that only 23% of a survey of faults in widely-used programs were of the sort generated by existing mutation systems. A testing technique that does not address the majority of faults may be less effective than one that does.

the possibility of low cost-effectiveness
: In this study, only 40% of the faults generated were judged to be typical of those made by programmers. This suggests that the cost-effectiveness may be low: little is gained by ruling out a fault that is unlikely to exist.

It is important to emphasize that these problems have neither been proven to exist nor disproven. The *coupling effect* [DeMillo78] hypothesizes that the first does not exist, that a test suite adequate to detect all simple faults will also detect most complex faults. It has been verified [Offutt89], but only for faults composed of multiple simple faults. [Marick90] reports that such compound faults are only 8% of those found in a sample of faults. (47% of the faults were faults of omitted code. 23% were other complex faults, containing both omitted code and changes to existing code. Other studies, such as [Basili84], [Glass81], and [Ostrand84], have shown similar proportions of omission faults.)

Similarly, the cost-effectiveness might be high despite a large proportion of unlikely faults. Test cases for typical faults might detect unlikely faults. If so, the unlikely mutants might require no additional tests.

Experiments to examine these potential problems are needed. In the meantime, a technique that partially avoids them appears possible. This technique is based on the observation that program faults are no more random than programs are. It has long been known that programmers typically program by composing *cliches*, well-understood computational patterns [Rich90]. There is evidence that novice programmers make cliched faults when attempting to implement those cliches [Johnson83], [Spohrer85]. If the same is true of professional programmers, a good testing strategy would be the following:

(1) Derive tests from the specification.

(2) Examine the program for cliches.

(3) For each of the cliches, select necessity conditions for its common faults and instrument the code to detect whether the cliche is exercised under those conditions.

(4) Simple necessity conditions like those used in this experiment would be used only for code not a part of any cliche.

(5) Create tests until all the conditions are satisfied.

Such an approach could be both more effective and more cost-effective than mutation testing. The cost is lowered if a cliche as a whole has a smaller number of necessity conditions than its component parts do. The effectiveness is raised if, first, the cliched necessity conditions suffice to detect faults of omission or other complex faults. For example, suppose one is writing a command line parser for commands of the form:

commandname -f *filename*

The -f is optional; if supplied, it requires a file name as an argument. A common programming error is to fail to check whether the *filename* is actually supplied when the -f option is given. In UNIX, the usual result is dereferencing a null pointer. A test case satisfying the necessity condition for this fault is

commandname -f

The necessity condition is easily satisfied, even though the exact form or location of the fault (an omitted test) is not known.

Effectiveness further requires that tests for the cliched faults also detect uncliched simple faults: an "inverse coupling effect". For example, tests for the command line parser's necessity conditions must discover wrong-variable-used faults.

Finally, effectiveness requires that the local effect of a fault usually persist until output (the weak mutation hypothesis). This study suggests that it does. In some cases, such as boolean cliches, simple instrumentation of necessity conditions may need to be augmented by other instrumentation that checks partial sufficiency conditions or produces execution traces for later analysis [Howden87]. Where even higher reliability is needed, test cases that satisfy the sufficiency condition may be generated, using techniques like those of RELAY [Richardson88] or Morell [Morell90]. However, this analysis is likely to be more difficult than for simple faults: cliches have gaps containing non–cliche text, often overlap with other cliches, and may have multiple exit points [Ning89].

The result of this study is that there is as yet no evidence against the technique. There is also no evidence for it. A catalog of cliched faults is being created, and instrumentation tools are being built. The design of an experiment to measure the technique's effectiveness and cost–effectiveness has begun.

7. Acknowledgements

Georgios Papagiannakopoulos checked for mistakes by independently classifying a subset of the faults; in doing so, he noticed that faults in boolean functions are more apt to be undetected.

REFERENCES

[Basili84]
V. Basili and D. Weiss "A Methodology for collecting valid software engineering data". *IEEE Transactions on Software Engineering,* vol. SE–10, pp. 728–738, November, 1984.

[Basili87]
V. Basili and R.W. Selby. "Comparing the Effectiveness of Software Testing Strategies". *IEEE Transactions on Software Engineering,* vol. SE–13, No. 12, pp. 1278–1296, December, 1987.

[Budd80]
Timothy Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. *Theoretical and Empirical Studies on Program Mutation to Test the Functional Correctness of Programs.* Technical Report GIT–ICS–80/01, Georgia Institute of Technology, 1980.

[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.

[Girgis86]
M.R. Girgis, M.R. Woodward. "An experimental comparison of the error exposing ability of program testing criteria". In *Proceedings of the Workshop on Software Testing Conference,* pp. 64–73, Banff, Canda, 1986.

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

[Hennell84]
M.A. Hennell, D. Hedley, and I.J. Riddell. "Assessing a Class of Software Tools". *Proceedings of the 7th International Conference on Software Engineering,* pp. 266–277, IEEE Press, 1984.

[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.

[Howden80]
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.

[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.

[McCabe76]
T. McCabe. "A Complexity Measure". *IEEE Transactions on Software Engineering*, vol. SE–12, No. 4, December, 1976.

[Morell83]
L.J. Morell. *A Theory of Error–Based Testing.* Ph.D. dissertation, University of Maryland, 1983.

[Morell90]
L.J. Morell. "A Theory of Fault–Based Testing". *IEEE Transactions on Software Engineering*, Vol. SE–16, No. 8, August 1990, pp. 844–857.

[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.

[Ning89]
J. Q. Ning. *A Knowledge–Based Approach to Automatic Program Analysis.* Ph.D. dissertation, University of Illinois, 1989.

[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.

[Ostrand84]
Thomas J. Ostrand and Elaine J. Weyuker. "Collecting and Categorizing Software Error Data in an Industrial Environment". *Journal of Systems and Software*, Vol. 4, 1984, pp. 289–300.

[Pearson04]
K. Pearson. *On the Theory of Contingency and Its Relation to Association and Normal Correlation.* London: Draper's Co. Memoirs, Biometric Series No. 1, 1904.

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

[Richardson88]
Debra J. Richardson and Margaret C. Thompson. "The RELAY model of error detection and its application". *Proceedings of the ACM SIGSOFT/IEEE Second Workshop on Software Testing, Analysis and Verification*, Banff, Canada, July 1988.

[SERC87]
The Mothra Software Testing Environment.

Software Engineering Research Center report SERC-TR-4-P, Purdue University, 1987.

[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.

[Woodward80]
M.R. Woodward, D. Hedley, and M.A. Hennell. "Experience with path analysis and testing of programs". *Transactions on Software Engineering*, vol. SE-6, No. 3, pp. 278-286, May, 1980.

[Woodward88]
M.R. Woodward and K. Halewood "From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues". *Proceedings of the ACM SIGSOFT/IEEE Second Workshop on Software Testing, Analysis and Verification*, Banff, Canada, July 1988.