

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

An Empirical Study of the Branch Coverage of Different Fault Classes

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER AND INFORMATION SCIENCE

by

Melissa S. Cline

September 1995

The thesis of Melissa S. Cline is approved:

Linda Werner

Charles McDowell

Joel Ferguson

Dean of Graduate Studies and Research

Contents

Abstract	vii
Acknowledgements	viii
1. Introduction	1
1.0.1 Structural Testing	1
1.0.2 Software Development Phases	2
1.0.3 Affected Branch Coverage	5
2. Prior Work	10
2.1 The Early History of Research in Software Testing	10
2.2 Comparisons of the Effectiveness of Structural Testing to Other Forms of Testing	12
2.3 Studies on Industrial Software	21
3. Description of the Experiment	30
3.1 Description of the Software	30
3.2 Measuring Branch Coverage with EXMAP	30
3.3 Collection of the Faults and Fault Class Assignment	31
3.4 Relating Branch Coverage to Faults	33
4. Results	36
4.1 Analysis of the Faults Observed	36
4.2 Affected Branch Coverage for All Classes of Faults	37
4.3 Contrasts Between Affected Branch Coverage on the Two Groups	41
5. Conclusions	43

5.1 Areas for Further Study	44
A. Glossary	46
B. Raw Data	49
References	53

List of Figures

1.1	Examples of Deriving Affected Branch Coverage	7
2.1	An illustration of the cyclomatic complexity of a module	11
2.2	Branch Coverage vs. Data Flow Coverage	15
3.1	Sample EXMAP Summary Report	31
3.2	Excerpt from a Sample EXMAP Annotated Listing	34
4.1	Number of Faults vs. Number of Affected Branches	37
4.2	Number of Faults vs. Affected Branch Coverage	38

List of Tables

1.1	Average vs. Overall Affected Branch Coverage	8
2.1	Summary on Comparative Testing Methods	22
2.2	Populations of Major Fault Classes	29
4.1	Fault Class vs. Number of Faults	36
4.2	Average Number of Affected Branches by Fault Class	37
4.3	Number of Faults vs. Overall Affected Branch Coverage	38
4.4	Fault Class by Overall Affected Branch Coverage — Older Group	39
4.5	Fault Class by Overall Affected Branch Coverage — Newer Group	39
4.6	Fault Class by Overall Affected Branch Coverage — Both Groups	39
4.7	Overall Coverage of Affected Branches by Fault Class	40
B.1	Raw Coverage Results — Older Group	50
B.2	Raw Coverage Results — Newer Group	52

An Empirical Study of the Branch Coverage of Different Fault Classes

Melissa S. Cline

ABSTRACT

The question “How much testing is enough?” has led many to structural testing methods. Much has been written about their fault detecting ability, but how does this vary by the class of fault?

This thesis introduces the term *Affected Branch Coverage*. An affected branch is a branch which had to be modified in order to fix a fault. Affected Branch Coverage describes the percentage of affected branches that had been exercised in testing. The study was done on a leading on-line transaction processing product, analyzing ninety eight field errors.

The specific questions addressed are:

- Which classes of faults are most commonly observed?
- Which fault classes can be associated with covered code and which with uncovered code?
- Is affected branch coverage related to the maturity of the software?

Our results show that whether or not a fault would appear in covered code depends strongly on the fault class. While this was true in both newer and older code, it was more vivid in newer code. Overall, we found that affected branch coverage was slightly less than 50%. We observed that increasing branch coverage would offer little benefit for additional fault detection in certain fault classes, and approximately two-thirds of the faults studied were from these fault classes. Overall, this suggests that increasing branch coverage would offer only limited gains in fault detection.

Acknowledgements

This work was partially supported by the International Business Machines Corporation grant # STL93215.

This paper is the product of the work of a number of people, both at the University of California and at IBM Santa Teresa. It was largely due to Jerry Crane, a man with a vision, that these two groups of people came together.

This project would not have succeeded without the support of a number of people at Santa Teresa. The EXMAP output is the product of Frank Nargie, and the many evenings and weekends he spent in the lab. Jenny Chu and Barbara Wade devoted many hours to removing administrative roadblocks and seeing that the project had the necessary resources to run smoothly. Jim Lines never lost sight of the problems to be solved, and without his efforts nothing would have been accomplished.

At the University of California, much of the energy behind this project came from the students in CMPE276, the graduate software engineering class taught by Linda Werner. Their ideas and enthusiasm went a long way for laying the groundwork on this project, locating the relevant prior work, and asking the relevant questions early on. Linda Werner was always there with a cool head, an objective perspective, and offers to help with the work.

My working with Linda Werner, and my involvement in this project, is thanks entirely to Lynne Sheehan. Had it not been for Lynne's caring and her administrative genius, I would not be writing this thesis today.

1. Introduction

The goal of software testing is to determine if the software is correct by executing it. There are many techniques for software fault removal, such as inspections, walkthroughs, and reviews, but testing refers to the methods based on executing the software.

There are both formal and informal software testing methods. Informal software testing typically involves a person sitting in front of the keyboard pounding keys. While this can be very productive in revealing faults, informal testing alone is not considered adequate: it is difficult to ensure that all of the functionality of the software was tested [Beizer, 1990]. So most often, large software products are tested using formal methods.

Formal software testing involves one execution of the software with specified inputs and under specified running conditions, and comparing the results with expected results. A single tuple of inputs, conditions, and expected results is called a test case or a test. The testing of a software product involves executing numerous tests.

1.0.1 Structural Testing

This work focuses on structural software testing, which can be viewed as testing with two goals. The first goal is to determine if the software is working correctly. The second goal is to measure how much of the software has been executed in testing. Structural testing is driven by the principle that the thoroughness of testing is proportional to the amount of the software executed under test. Typically, the amount of the code executed is measured using special software testing tools. These tools instrument the code, keep a running tally of which sections of the code have been executed, and output a percentage of the sections executed versus all sections. This percentage is referred to as coverage. 100% coverage is the ideal for all forms of structural testing.

There are different forms of structural testing which use different granularities of sections. Some of the more common are statement coverage, procedure coverage, and branch coverage. Additional forms of structural testing are described in the glossary, appendix A. Statement

coverage describes what percentage of the source code statements were executed under test. Procedure coverage describes what percentage of the procedures were called under test. Obviously, statement coverage is more thorough than procedure coverage: if you have executed all of your statements, you have also executed all of your procedures.

On the surface, 100% statement coverage would appear to be the most thorough form of structural testing. However, you can do better. Consider the case of an *if* statement with a *then* clause but no *else* clause. 100% statement coverage will ensure that the *then* clause is executed. But suppose the software had an error which occurred whenever this *if* statement was executed with a false outcome — one which did not result in executing the *then* clause? Statement coverage does not ensure that this error would occur under test. While this might seem overly picky, conditional branch instructions represent 10% - 17% of all instructions in popular software benchmarks [Hennessy and Patterson, 1990], and control flow faults have been measured at 12.8% of all software faults [Beizer, 1990]. The structural testing method which will help find these faults is branch testing.

The goal of branch testing is to test both possible outcomes of all branchpoints. A branchpoint is defined as a point at which a decision is made and where based on the decision, the flow of execution takes one of two or more logical paths. Each one of these logical paths is referred to as a branch. Examples of branches include the bodies of loops and the *then* and *else* clauses that follow *if* statements. All source code statements are either branchpoints or sequential statements on branches. Therefore, 100% branch coverage is considered stronger than 100% statement coverage.

1.0.2 Software Development Phases

Coverage goals are just one way in which testing methods differ. Just as there are different steps in software development, there are different steps in software testing, with different methods employed at each step. Each method is effective at finding different types of faults, and a typical software development plan will employ multiple testing methods before release. The earlier a fault is found, the less expensive it is to fix [Pressman, 1992]. Therefore,

each testing method is employed as early as possible. Here, I will briefly describe a typical software development plan with the corresponding methods of fault removal. These methods are also described in appendix A.

Software development typically begins with writing the software requirements specification. This specification describes in precise terms what the software is to do. The fault removal method employed here is requirements review, where the reviewers look for holes, contradictions, or ambiguities in the requirements.

The next development phase is software design. The goal of this phase is to plan exactly how the software will execute, including the scope and requirements of each module. The design is verified with a design inspection. The goals of the design inspection are to verify that the design is complete and correct and that it supports the software requirements.

After the design has been verified, the programmers begin coding the modules. As each module is completed, unit testing is performed on it. Unit testing involves testing each module in a stand-alone fashion. It is usually done by the software developers, and is usually white-box testing: testing done using knowledge of the internals of the code. In contrast, black-box testing is done with no knowledge of the internals — only by comparing actual outputs with expected outputs.

Typically, test harnesses are written as unit testing aids. These test harnesses are simple main routines which do nothing but call the module under test. The goal of unit testing is to ensure that each module meets its design requirements perfectly. Internal consistency checks such as checks for invalid inputs are easily tested at the unit level: the test harness can simply call the module with invalid inputs. In contrast, when the full program has been built and when the module is being called from other modules in the program, it should not be possible to call the module with invalid inputs. Structural testing is mostly done in unit testing, because of the ease of testing the internal consistency checks.

In addition to unit testing, modules are often verified using code reviews. During a code review, a number of software developers will sit down and read the module very carefully. Since unit testing is typically performed by the programmer who writes the module, code

reviews provide a safeguard against that programmer overlooking something in both the coding and the unit testing.

As groups of related modules become ready, they are linked together and tested using integration testing. Integration testing is similar to unit testing: it is done at a white-box level, and typically with the aid of some testing harnesses. The goal of integration testing is to find inconsistencies in the interfaces between related modules.

When all related modules are ready and integrated, functional testing and system testing begin. Functional testing ensures that all of the required operations are performed correctly. Functional tests are typically black-box: the tester has no knowledge of the internals of the software and is only concentrating on its external operation. Functional testing may be performed at the system level, when all of the software has been completed and is linked together. But to speed up software development, the software product is often divided into functionally disjoint subsystems. While the last subsystems are being completed, functional testing is performed on the others.

When the entirety of the software is ready, system testing begins. System testing ensures that the software works correctly as one whole entity, and that all performance requirements are met. System testing is a form of black-box testing. Functional testing and system testing are sometimes combined into one phase.

Finally, starting with version 2 of the software, the maintenance and enhancement phase begins. This phase contains all prior steps, and adds regression testing. Regression testing is typically performed after all other testing phases. While the other phases concentrate on the new functionality, regression testing ensures the integrity of old functionality. Regression tests are black-box system level tests, and can be thought of as functional and system tests that relate only to the functionality of prior releases.

As mentioned earlier, structural testing is most often done at the unit level. But sometimes, coverage is measured during system level testing to get an approximation of testing thoroughness.

1.0.3 Affected Branch Coverage

Increasing branch coverage is a widely accepted method of increasing the effectiveness of testing. While branch coverage is most often measured in unit testing, in recent years it has gained popularity as a measure of various forms of system-level testing.

Prior work describes a strong, direct relationship between statement coverage and fault detection in functional test [Piwowarski *et al.*, 1993]. It seems reasonable to expect that the relationship between branch coverage and fault detection is at least as strong.

However, any form of structural testing has a limited effectiveness at fault detection. Even when testing with a criterion of 100% coverage, structural testing is not likely to reveal more than half of the faults in a body of code [Basili and Selby, 1987] [Girgis and Woodward, 1986] [Selby, 1986].

Additionally, high code coverage can be a very expensive goal. 100% code coverage may not be feasible outside of unit test for a number of factors including code handling “impossible” error conditions, dead code, hooks for new functionality, and code requiring special hardware. For larger systems, a system level coverage of greater than 50% might not be attainable [Beizer, 1990]. High coverage is also a goal that must be pursued deliberately — when coverage on a software product is measured for the first time, those involved with the testing are often surprised to learn how low the coverage really is [Grady, 1993] [Piwowarski *et al.*, 1993]. When attempting to increase coverage, testing teams will sometimes become overly focused on increasing coverage and will lose sight of the underlying quality goals [Su and Ritter, 1991].

When we set out to increase branch coverage, it is important to know exactly what gains can be expected. One factor in this equation is what classes of faults are detected effectively by branch testing. Prior work suggests that the fault-detecting ability of branch testing is not uniform across fault classes.

One prior study showed that while structural testing revealed more faults than 2-version voting, code reading, assertions, or static analysis, there were classes of faults for which it

was ineffective. These classes included missing checks, parameter reversal, substitution, and calculation faults [Shimeall and Leveson, 1991]. Another study showed that structural testing in general is more effective in finding faults affecting the logical flow of the program than faults affecting the data state, and that branch testing in particular is not effective in detecting the wrong arithmetic operator or a statement wrongly placed in a logical expression [Girgis and Woodward, 1986]. A third study showed statement coverage to be more effective at detecting faults of involving the wrong operation than faults involving a missing operation, and weakest at detecting cosmetic, interface, and data faults [Basili and Selby, 1987].

Therefore, we decided to examine a large, industrial software product and the testing that is performed on it. By looking at errors reported from the field, and whether or not their underlying faults had been covered in testing, we have acquired a snapshot of the faults that slip by branch testing.

Many software faults require a fix in more than one place. To look at whether or not the fault had been covered, you must look at the coverage in all places that had to be fixed. In order to do this quantitatively, we defined some new terms. If any statement on a branch had to be modified to fix a fault, we call that branch an *affected branch*. If a branchpoint had to be modified, then implicitly both branches emanating from it are affected branches. We use the term *affected branch coverage* to describe the percentage of all the branches affected by a fault which had been covered in testing. This is illustrated in figure 1.1.

Affected branch coverage tells us to what extent a single fault was covered. But what about all of the faults? Here, we look at two numbers. First, the *overall affected branch coverage* is determined by adding up the number of covered affected branches for all of the faults and dividing that by the total number of affected branches. Additionally, the *average affected branch coverage* is computed by determining the affected branch coverage for each fault, and averaging all the individual affected branch coverages. Average and overall affected branch coverage are illustrated in table 1.1.

The average affected branch coverage weighs all faults evenly: faults affecting one branch

Examples of Affected Branch Coverage

Below are some sample code fragments. Lines marked with * are lines which had to be changed to correct the fault. In the leftmost column, Y marks statements which had been covered in testing, and N marks statements which had not been covered.

Example 1:

Y	if (x > 0) {	
Y	a = 1;	Number of Affected Branches: 1
Y	b = 1;	Number of Affected Branches Covered: 1
Y	c = foo2(a,b); *	Affected Branch Coverage: 1/1 = 100%
	}	

Example 2:

Y	if (x > 0) {	
Y	a = 1;	
Y	b = 1;	
Y	c = foo2(a,b); *	Number of Affected Branches: 2
	}	Number of Affected Branches Covered: 1
Y	if (y > 0) {	
N	d = foo3(y,1); *	Affected Branch Coverage: 1/2 = 50%
N	e = foo3(y,2); *	
	}	

Example 3:

Y	if (x > 0) { *	
Y	a = 1;	
Y	b = 1;	Number of Affected Branches: 2
	} else {	Number of Affected Branches Covered: 1
N	a = 2;	
N	b = 2;	Affected Branch Coverage: 1/2 = 50%
	}	

Figure 1.1: Examples of Deriving Affected Branch Coverage

are weighed the same as faults affecting multiple branches. Average affected branch coverage depicts the fault coverage as if there is a uniform cost to each fault, independent of how much code the fault affects. This is useful for equations with a fixed cost per fault, such as the one used by IBM to estimate support costs. In contrast, overall affected branch coverage weighs faults in proportion to their size. This is useful for equations in which each faulty branch should carry the same weight, regardless of how many branches are affected by a fault. So, this is useful for describing whether certain classes of faults tend to be covered or not. The overall affected branch coverage is especially interesting when compared to the branch coverage, the percentage of all branches (faulty and not) which had been covered in testing.

Fault #	# Affected Branches	# Covered	Affected Branch Coverage
1	10	2	20%
2	1	1	100%
3	2	0	0%
4	4	3	75%
5	1	0	0%
Total # Affected Branches: 18			
Total # Covered: 6			
Overall Affected Branch Coverage: 33.3%			
Average Affected Branch Coverage: 39.0%			

Table 1.1: Average vs. Overall Affected Branch Coverage

The average affected branch coverage is the average of the five numbers in the affected branch coverage column. The overall affected branch coverage is the percentage of the affected branches that were covered.

By comparing the overall affected branch coverage to the branch coverage of the software, we can draw some interesting conclusions about how to improve the testing. First, suppose that affected branch coverage is significantly lower than the branch coverage. That would imply that most of the faulty branches are in the code not currently covered. In that case, increasing coverage could be a very productive thing, because it could help find some of the faults in that body of untested code.

In contrast, suppose that affected branch coverage is significantly higher than branch coverage. That would tell us that most of the faulty branches are covered already: increasing coverage would offer only limited gains. It would probably be more worthwhile to consider the faults occurring in the covered code, and determine how to modify the testing to detect more of them.

Finally, as mentioned earlier, certain classes of faults are detected most easily through certain forms of testing. Branch testing is most effective at finding faults that perturb the logical flow of the software. To get a good view of the thoroughness of the testing, it is useful to divide the faults into fault classes, and to look at the overall affected branch coverage for the classes where we would expect branch coverage to do well.

For the classes of faults, we chose to use the same taxonomy used in previous studies of database and on-line transaction processing (OLTP) systems [Sullivan and Chillarege,

1992]. This breakdown focuses on software implementation faults rather than faults earlier in the lifecycle. The fault classes used are listed in 3.3

For each of these categories, we investigate the relationship between affected branch coverage and fault incidence. We have found that whether or not a fault will be in covered code depends heavily on the fault class. For instance, the overall affected branch coverage of the data fault class is only 30.4% while the average affected branch coverage of interface faults is 70.2%.

Finally, we will contrast the results of our two sample groups. The first is an older group, containing modules with a high historical error rate. The second is a newer group, produced by a team known for their methodical engineering techniques. We have found that for the older group, affected branch coverage is lower than branch coverage. For the newer group, affected branch coverage is higher than branch coverage. So, the newer group shows a greater density of faults in covered code than the older group. This contrast might point to the fact that the software in the newer group is still maturing. The distribution of faults among the classes was significantly different for the two groups. In the newer group, there was a smaller population of "undefined state" faults, perhaps implying that the engineering techniques used on the newer group has been successful at reducing a major population of faults.

2. Prior Work

2.1 The Early History of Research in Software Testing

Software testing is a discipline in which the practice leads the theory [Gelperin and Hetzel, 1988]. Software testing is as old as software development, but software test engineering has only emerged as a discipline in the last twenty years. Consequently, much of the research in software testing aims to analyze the effectiveness of widely-used testing practices.

In the earliest days of software development, programs were very simple and the complexity of the system was in the hardware. The original function of software testing was to aid in hardware debugging. In the sixties and early seventies, as programs increased in size and complexity, the substantial cost of software faults was recognized. At this time, testing ceased to be viewed as strictly a debugging activity and grew into a technique of ensuring the correct operation of software.

The new role of software testing introduced software test engineering as a career path distinct from software development. Software testers were given new challenges: to make their testing more thorough, faster, and more cost-effective. The first formal conference in software testing was held in 1972. In 1979, Glenford Meyers published his book, *The Art of Software Testing*. Following this, software testing books started appearing at the rate of about two per year.

During the late seventies, software testing articles began appearing in the software engineering journals at an increasing rate. One of the more notable early articles was published in 1976 by Thomas McCabe [McCabe, 1976]. McCabe showed that the effort required to test a module was proportional to its cyclomatic complexity, and defined the cyclomatic complexity of a module as follows:

$$cc = l - n + 2$$

where l is the number of links in a flowgraph representation of the module, n is the number of nodes in this flowgraph, and cc is the cyclomatic complexity. This is illustrated

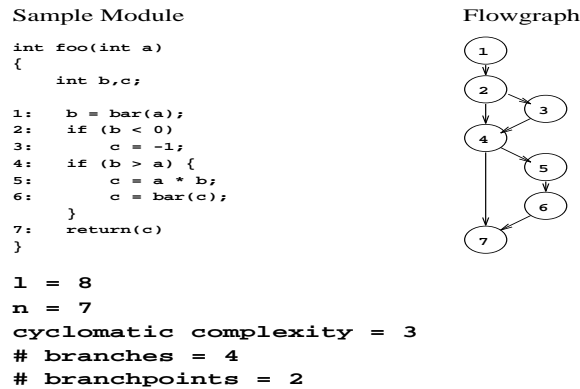


Figure 2.1: An illustration of the cyclomatic complexity of a module

in figure 2.1. When one considers these quantities, it is easy to see that a group of sequential statements can be combined into a single node without affecting the cyclomatic complexity. Considering this further, two nodes cannot be collapsed into one when the parent node has more than one path emanating from it. So, when there is a branchpoint at the parent node, the complexity will increase by one. Ultimately, assuming a programming language that uses binary decisions, the cyclomatic complexity is equal to the number of branchpoints plus one, with a branchpoint being any point at which the flow of execution could proceed down more than one path.

Cyclomatic complexity had a significant advantage over prior complexity metrics: it is very easy to measure. The metrics-hungry world of software engineering embraced cyclomatic complexity with enthusiasm, and it was soon shown that the cyclomatic complexity of a module is not just related to the time required to test but is also related to the fault density and the time required to comprehend. Thus, measurement of cyclomatic complexity became a common practice, and its use helped promote branch coverage to a standard measure of testing thoroughness.

2.2 Comparisons of the Effectiveness of Structural Testing to Other Forms of Testing

As stated earlier, software testing is a discipline in which the practice leads the theory. At the time McCabe wrote on cyclomatic complexity, structural testing and coverage measurement had become a common practice in industry. However, structural testing was done on an instinctive basis, with the philosophy that “testing cannot reveal the faults that have not been executed”. At this time, there had been no major studies on the effectiveness of structural testing.

The mid to late eighties proved to be a renaissance for structural testing research. A number of articles set out to quantify the effectiveness of structural testing and to relate it to other testing methods. The first of these, written by James Ramsey and Victor Basili, set out to measure the statement coverage of functional testing [Ramsey and Basili, 1985].

Their subject was a commercial Fortran program with 68 procedures and four thousand executable statements and eight known faults, instrumented to collect data on statement coverage and *procedure coverage*. Procedure coverage is a measure of what percent of the procedures had been executed during testing. Procedure coverage does not measure how thoroughly a procedure has been executed: it only measures whether or not the procedure has been called at least once. Statement coverage and procedure coverage were measured while the software was executed with a small number of functional test cases and a larger number of operational usage cases. The coverage for functional testing and operational use was then compared. Then, this coverage information was related to the location of the eight faults.

Each of the functional test cases covered between 21% and 46% of the statements. Overall, there was a strong correlation between the statements executed in functional testing and those executed in operational usage. The statement coverage from functional testing was 57%. Almost all of the statements that were executed under functional test cases were also executed under the operational usage cases, although operational usage exercised an additional 8.4% of the statements. The classes of statements most often executed in

operational usage and not in functional test were reads and writes, which is interesting given the importance of reads and writes in debugging.

Intuitively, the rate of statement coverage would not increase linearly with added testing: certain portions of the code would be common to all test cases while other portions of the code would be very specific to one functional area. After all, every test case must execute the main routines. Indeed, the researchers found that half of the procedures were executed by 90% of the test cases. Plotting the rate of coverage against test execution time, they found the following relationship:

$$m(t) = a(1 - e^{-bt})$$

where t represents test execution time, b is a constant greater than or equal to one, a is the total number of statements, and $m(t)$ is the statement coverage at time t .

Additionally, the researchers found about half of the faults in the more heavily covered procedures, and found that while procedure coverage is apparently independent of time to isolate, procedure coverage is directly related to time to understand the procedure and implement the fix. However, no worthwhile conclusions can be drawn from eight faults.

The second major study in comparative testing methods was published in 1986 by Girgis and Woodward [Girgis and Woodward, 1986]. They set out to compare the error detection ability of structural testing, weak mutation testing, and data flow testing [Girgis and Woodward, 1986].

In mutation testing, the effectiveness of a test suite is measured by “mutating” the software — injecting known faults, and measuring the effectiveness of the test suite in detecting the mutants. Once the test suite has been found effective in detecting the known mutants, it is then used to test the non-mutated software. Weak mutation testing is a variation which focuses on five specific mutations — wrong variable referenced, wrong variable assigned, incorrect arithmetic expression, incorrect relational expression, and incorrect boolean expression. The goal of weak mutation testing is to be able to detect each of these mutants with at least one test case [Howden, 1982]. Mutation testing can detect faults that branch testing will miss. However, a test which can detect one mutation

cannot necessarily detect all mutations. This is the weakness in mutation testing. In weak mutation testing, this weakness is compounded because the tests are written to find a smaller set of mutations.

Data flow testing is another form of structural testing, and is a refinement of path testing. Intuitively, branch testing is stronger than statement testing because 100% statement coverage must be achieved in order to achieve 100% branch coverage, but not vice-versa. Likewise, path testing, in which the goal is to execute all combinations of all branches, is stronger than branch testing because 100% branch coverage is a subset of 100% path coverage. The most common criticism of path testing is that the number of test cases required for full path coverage grows exponentially with the number of branches, and that many of those test cases are not useful in detecting faults. Intuitively, when testing a module that contains two unrelated branches, testing all combinations of the branches offers little gain over just testing each branch once. This criticism led to the introduction of data flow testing, which offers the thoroughness of path testing without the meaningless or redundant test cases.

The goal of data flow testing is not to test all paths, but instead to test each path over which a variable is *active*. A variable is defined to be active on the paths from its first assignment to its last use. So, if a procedure operates on variable x , and then contains some unrelated code in which it operates on variable y which is independent of x , the branches relating to variable y are of no concern for the data flow coverage of x . Likewise, the branches relating to x do not affect the data flow coverage of y . In this way, data flow testing improves on path testing by covering only the paths on which a fault is likely to be revealed. These concepts are illustrated in figure 2.2. Notice in this figure that even though ten data flow cases are shown, only five of them are unique. In addition, there are two infeasible test cases not shown, for a total of seven. In contrast, 100% path coverage would have required nine test cases.

Girgis and Woodward performed their study on five small Fortran programs, selected from Fortran textbooks. The programs were seeded with faults using an automated tool,

Sample Procedure

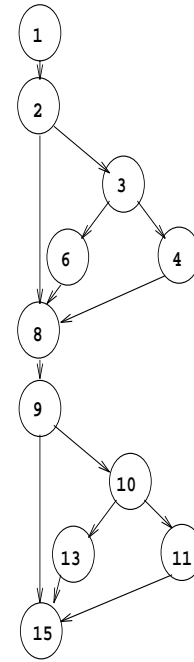
```

int getmean(FILE *outfile, int *list)
{
    float mean, std;

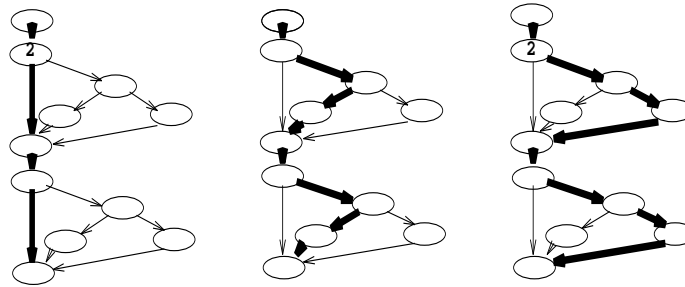
1:  mean = getmean(list);
2:  if (mean == 0.0) {
3:      if (outfile != NULL)
4:          fprintf(outfile, "Population centered\n");
5:      else
6:          printf("Population centered\n");
7:  }
8:  std = getstd(list, mean);
9:  if (mean == 0.0 && std == 1.0) {
10:     if (outfile != NULL)
11:         fprintf(outfile, "Population normalized\n");
12:     else
13:         printf("Population normalized\n");
14: }
15: return(std);
}

```

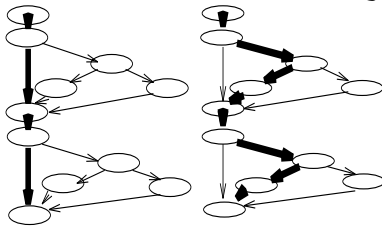
Flowgraph



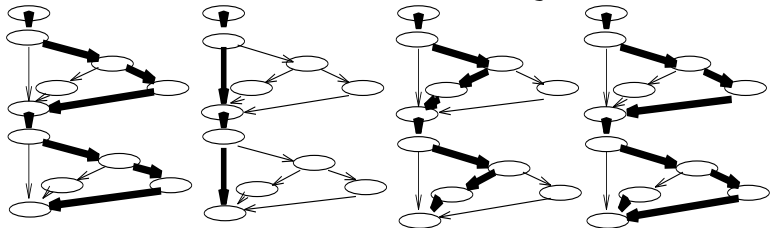
100% Branch Coverage



100% Data Flow Coverage of mean



100% Data Flow Coverage of std



100% Data Flow Coverage of outfile

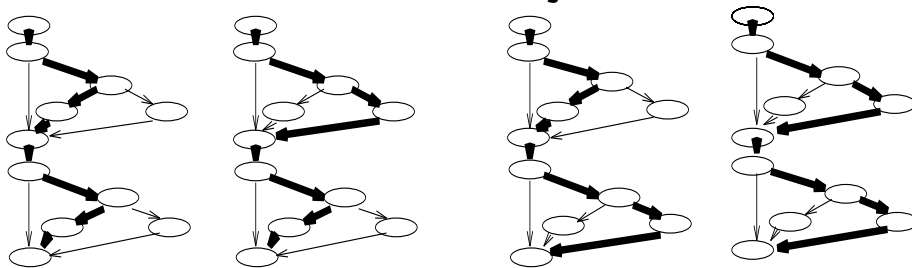


Figure 2.2: Branch Coverage vs. Data Flow Coverage

and the execution of the seeded program was compared to that of the non-seeded program. When the outcomes differed, the researchers claimed that the testing had uncovered one of the faults. Because the faults were injected by an automated tool, the researchers had no knowledge of their location or nature. Therefore, the testing could be considered unbiased.

These faults were divided into two major classes: domain faults, which perturb the logical flow of the program, and computation faults, which do not directly affect the logical flow but directly affect the data state. In general, computation faults are faults in assignment statements or i/o statements, and domain faults are faults in conditional statements. For example, if a pointer is assigned to the wrong variable, that is a computation fault. If a pointer is compared to the wrong variable, that is a domain fault.

In addition to these major classes, the faults were divided into subclasses. The subclasses used for the computational faults were wrong variable assigned, wrong arithmetic operation, wrong constant, wrong variable referenced, statement wrongly placed, and missing computation. The subclasses for the domain faults were wrong arithmetic operation, statement wrongly placed, wrong variable, wrong constant, and wrong relational operator.

After seeding the programs with these faults, the researchers tested them with criteria of 100% mutation coverage, 100% structural coverage, and 100% data flow coverage. A number of structural coverage methodologies were used including statement testing and branch testing.

Overall, structural testing was found to be more effective than either data flow testing or mutation testing. Branch testing in particular caught 79% of the domain faults and 20% of the computational faults. This suggested that branch testing is more effective against faults that perturb a program's domain, or logical state, than faults that perturb the data state.

In the subclasses of faults detected, there was more revealed about the strengths and weaknesses of branch testing as a fault detecting mechanism. While branch testing detected 100% of the wrong variable and wrong constant domain faults, and 80% of the wrong relational operator domain faults, it detected only 13 to 25% of the computational faults

and did not detect any of the wrong arithmetic operator or statement wrongly placed domain faults.

Basili and Selby performed a similar study in 1987, comparing the effectiveness of a number of fault-detection strategies [Basili and Selby, 1987]. The strategies they studied were functional testing using equivalence partitioning and boundary value analysis, structural testing using 100% statement coverage, and code reading using stepwise refinement.

Equivalence partitioning is a methodology of great importance when testing software with continuous-valued inputs. It is assumed that at some level, there are sets of inputs that are treated in the same way. For example, a function that computes the calendar date based on the Julian date will operate one way for all dates in a leap year and another way for all dates not in a leap year. So, all leap years will fall in one equivalence class, while all other years will fall into another equivalence class. The goal of equivalence partitioning is to see that testing covers all equivalence classes, and the assumption is that multiple test cases which exercise the same equivalence class are redundant.

Loops are known to be one of the trickier constructs in software development. All software developers, no matter how experienced they are, have had their share of “off by 1 errors”. These errors frequently occur in the initialization or termination of a loop. Similarly, the difference between $>$ and \geq can make the difference between a robust program and one that crashes when presented with certain “magic numbers”. Boundary value analysis recognizes that software can be the most fragile at the boundaries of logical conditions, and sees that such inputs are included in the testing.

Reading code can be a very arduous task. This is especially true for the reader who is setting out to read an entire program, as opposed to one who is reading a few functions at a time. Stepwise refinement is a system for helping the reader keep the code in perspective. The reader starts at the highest level, usually the main program. As the readers continue to read the program, detail is slowly added by the reading of each procedure. In order to make the code reading a better approximation of the worst case, all comments were removed from

the code.

The researchers performed their work on a set of programs from different functional areas, chosen to be representative of industry. The programs varied in application, though unfortunately they were all very small programs. The largest program studied contained less than 150 executable statements. This size distribution ultimately diminished the effectiveness of the study, because the programs were so small that functional testing achieved almost 100% code coverage. The subjects were also chosen to be representative of industry. A fair majority were undergraduates studying software engineering. Others were professional software developers from the Computer Sciences Corporation. The faults were a combination of seeded faults and known faults in the programs.

For each testing method, the researchers collected how many faults were detected plus how many faults were observable but not detected. Not all observable faults are necessarily detected. The test protocol specifies a set of inputs and execution conditions, and a set of expected results. Sometimes the input domain is so large or the inputs so difficult to reproduce that a specific result cannot be given, only an approximation of what the result should be like. An example of this is anything that is derived from the system time if the tester cannot set the system time before testing. In these cases, results that are incorrect but only slightly off can slip by the tester quite easily. Even seemingly obvious results can slip by a tester: if the result is a number displayed on the screen with accompanying text, the tester might focus on the value of the number and fail to notice typographical errors in the text.

Overall, 49.95% of the faults were detected, while a total of 68% were observable. Code reading and functional testing were found to be about equally effective, detecting 54.1% and 54.6% respectively. Structural testing, on the other hand, detected only 41.2% of all faults.

Structural testing was found to be slightly better in faults of commission than faults of omission (44.3% vs. 39.2%). The distinction between the two categories is that faults of omission involve forgetting to do something while faults of commission involve doing

the wrong thing. Additionally, structural testing proved better at finding faults in the computation (58.8%), control (48.8%), initialization (46.2%), and worse at data (26.8%), interface (29.4%), and cosmetic (7.7%) faults.

As mentioned earlier, the study also recorded the number of faults observable but not detected. This offers a picture of the number of faults that could be detected with optimal analysis of the test results. The structural testers failed to report 19.9% of the observable faults. The faults most frequently not observed were cosmetic faults — 85.7% were observable but not detected. The test analysis proved to be most thorough on data faults, with only 7.5% observable but not detected. For the remainder of the fault classes, approximately 20% of the faults were observable but not detected. Faults of omission and faults of commission were missed at approximately the same rate (21.3% vs. 20.1%). Overall, an additional 18.1% of the faults were observable in functional testing, in contrast. By definition, 100% of all faults are considered observable in code reading, leaving 45.9% of the faults observable but not detected by this method.

The last major work in the comparative effectiveness of fault detection systems was published in 1991 by Shimeall and Leveson [Shimeall and Leveson, 1991]. Using a set of programs written from a single specification, each program containing approximately 2500 statements, they set out to compare the fault detection ability of code reading, static data-flow analysis, run-time assertions, multiversion voting, and functionally-based structural testing, using all-p-uses.

A certain class of faults can be detected by an automated tool without either running or reading the software. The classic example of this is in the C language, using the assignment operator `=` in a logical expression rather than the logical equality operator `==`. Another example is operating on a variable that has not been assigned a quantity. Detecting these faults is the goal of static data-flow analysis. While static data-flow analysis alone is not an effective fault detection scheme, it is a powerful component of a scheme because the faults are detected with virtually no human effort.

Run-time assertions are a system to ensure the reliability of the software by having the

software verify its own accuracy. For example, a procedure could use run-time assertions to ensure that its inputs are within the expected range. When an assertion fails, the action taken depends on the system. Typically, the system terminates with a message stating where the assertion failure occurred.

In multiversion voting, a number of independent software development teams set out to build the same program. The assumption is that the two teams will not make most faults in the same place. By observing when the operation of the programs begin to differ, the tester can detect faults in any one of the programs. The assumption that two teams will not make the same faults in the same place has since been discounted: it has been found that different programmers starting with the same specification *will* often make the same mistakes in the same places. Consequently, multiversion voting has fallen from favor as a method of fault elimination.

All-p-uses is a form of data flow testing. Rather than testing all paths from all definitions to all uses of a variable, all-p-uses only requires testing all paths from all definitions to all predicate uses, or uses in a logical expression *with no re-definitions along the way*. If there is no way to get from one definition to one predicate use without executing a second definition of the same variable, the path from the first definition to the predicate is not required for all-p-uses.

The participants in this study were upper-division software engineering students. In order to keep the fault detection unbiased, one group performed all the design and implementation. Another group performed all the testing. Student arbitrators made the final determination on what behavior anomalies could be considered errors. While the testing was performed, for each fault detection method the participants collected the number of faults detected and the time required to detect those faults.

Structural testing took by far the most human hours to execute, in spite of the fact that the testers were using an automated testing tool. It took 373 hours, compared to 36 hours for code reading, 6 hours for multiversion voting, and 1 hour for static analysis. Structural testing also took the second longest in terms of computer hours. However, it proved to be

the most effective method of fault detection.

Naturally, there were classes of faults against which structural testing was not effective. These were faults involving the incorrect mathematical formula, the wrong variable used, an exception case not supported, and mis-ordering of the parameters in a function call.

It was observed that when structural testing was effective against faults, the test cases which proved most effective were those using unusual data or exercising special cases. Additionally, it was observed that the areas in which structural testing failed tended to be the same areas in which multiversion voting failed also, implying that the sections of a program which are difficult for software engineers to design and program are also difficult for software test engineers to write tests for.

The studies comparing the effectiveness of structural testing to other methods are summarized in table 2.1.

2.3 Studies on Industrial Software

While the work on the effectiveness of testing methods proved quite interesting to some, the industrial community remained skeptical. “Inexperienced programmers working with toy programs!”, echoed the concerns of industrial software practitioners. It was true that much of the existing software testing research had been done using very small programs and student programmers. It has been shown that the development of large software systems takes on a different dynamic and takes far more time than that of small software systems [Putnam and Myers, 1992]. It seemed reasonable to expect that software testing would reveal a similar dynamic: changing in some fundamental character as the size of the system increased.

Naturally, it is easiest for a university professor to perform research using small programs and student programmers. The programmers are available, and the experiment can be controlled. Industrial-based research is fraught with pitfalls such as politics, economic conditions, and deadlines, all of which can change quite suddenly and thwart any research efforts that may be going on. Nonetheless, the mid-eighties did yield some interesting

Study	Coverage Level	Compared to Other Methods	Effectiveness
Girgis & Woodward	100% Branch Coverage	Better than Data Flow Testing Better than Mutation Testing	79% Domain
			100% Wrong Variable 100% Wrong Constant 80% Wrong Relational 0% Wrong Arithmetic 0% Stmt. Wrongly Placed 20% Computation
Basili & Selby	100% Statement Coverage	Worse than Code Reading Worse than Functional Testing	41.2% Overall
			44.3% Commission 39.2% Omission 58.8% Computation 48.8% Control 46.2% Initialization 26.8% Data 29.4% Interface 7.7% Cosmetic
Leveson & Shimeall	All-P-Uses Data Flow Coverage	Better than Code Reading Better than Assertions Better than Static Analysis Better than 2-Version Voting	Good: Missing Branches Good: Wrong Relational Good: Missing Branch Good: Stmt Wrongly Placed Poor: Wrong Arithmetic Poor: Wrong Variable Poor: Wrong Parameters

Table 2.1: Summary on Comparative Testing Methods

publications on industrial software testing.

In 1984, Ostrand and Weyuker published an empirical study on industrial fault data [Ostrand and Weyuker, 1984]. They studied 173 faults on a commercial text editor, detected during unit testing, functional testing, or system testing. For each fault, they asked the programmer for the cause, the phase detected, and the time-to-fix. After studying the fault reports, they assigned the faults to classes. Like the authors in the studies in fault detection, they made a distinction between faults of omission and faults of commission.

Approximately 56% of the faults studied had been detected in unit or functional testing. Of these, the major classes were data definition faults (32%) and data handling faults (22%), decision and processing faults (18%), and decision faults (18%).

Faults of omission represented the vast majority of these faults, including 97% of the decision plus processing faults and 65% of the decision faults. In contrast, 68% of the data definition faults stemmed from incorrect code. Most of the faults were quick fixes once detected — 77% were isolated in less than an hour, and 71% were fixed in less than an hour.

The largest single cause of faults was programmer error, accounting for 63% of the total. Other major causes were unclear specifications (12%) and clerical (9%).

Also in 1984, Basili and Perricone published a similar study on the faults in the Software Engineering Lab at NASA [Basili and Perricone, 1984]. In this study, he collected fault data from thirty-three months of testing and operation of a 90,000 line Fortran system. For each fault, he collected data on the cause and the location, and assigned the fault to a fault class.

89% of the faults studied were well-localized, affecting only one module. There was a slightly smaller density of faults in the modules that had been reused from a previous system than those first written for the existing system. Overall, fault corrections resulted in 62% of the changes to the system — a figure which highlights the cost of faults in a large software system.

In contrast to the study by Ostrand and Weyuker, Basili found that a strong majority of faults were faults of commission (64%), with 35% faults of omission and 1% unclassifiable. The largest class of faults were interface faults (39%), followed by computation (19%), control (16%), data (14%), and initialization (11%).

The most surprising finding in this study was that there was a slightly larger concentration of faults in the smaller modules. This finding goes against the intuition that good modularity reduces faults, and was attributed to the large population of interface faults.

During the late eighties and early nineties, software quality assumed a new importance in the software engineering world. For many years, thorough and systematic testing had largely been the domain of government contractors. But during this time, the role of software grew

in financial and medical applications: arenas in which the cost of a defect can be extremely high. A few very expensive software errors occurred. One of the more notable faults caused a major outage in AT&T's nationwide telephone network and raised major questions about the system's reliability [Sims, 1990]. This error, which would ultimately cost AT&T billions of dollars in lost revenue, had its root in an omitted branching statement — a fault which could easily be detected with branch testing. Such errors, and the advent of the ISO-9000 standards, brought concerns of software quality to the commercial arena. Subsequently, this period saw a rise in publications based on measurement of industrial software quality.

In 1991, Ritter and Su published a report on their experience in testing the Motif user interface software [Su and Ritter, 1991]. With a schedule that allowed a mere seven months from the requirements specification to the release of a 100,000 line software product, the pressure was intense. The team knew that with such a schedule, the testing would have to be very efficient and thorough.

The testing team established some very respectable goals including coverage goals of 85% branch coverage, 100% procedural coverage, and 100% functional coverage. Additionally, they set out to create an automated test suite which would test new installations at these levels of coverage, *on any hardware platform supported*. The prevalence of the Motif interface today is a testimony to their efforts.

One pitfall they came across was the “branch coverage metric trap” — branch coverage is so easy to measure that it is difficult not to focus on it. The testers came to realize that when they set out to increase coverage, they would write mediocre tests. They would find themselves focusing on how to execute the uncovered branches rather than concentrating on the untested functionality that the branches represent. The final warning from the authors is that while branch coverage is very useful for measuring testing progress, it should not be used to write meaningless tests.

While 100% code coverage is a worthwhile goal, it can be very difficult to achieve, particularly in large, complex industrial systems. This fact led IBM researchers Paul Piwowarski, Mitsuru Ohba, and Joe Caruso to study and model the relationship

between fault detection and branch coverage in functional test, as seen in industrial software [Piwowarski *et al.*, 1993].

The first finding of these researchers was that when statement coverage was measured for the first time, testers tended to be surprised by how low it was. First time coverage rates of 50% to 60% were typical, while testers often had estimated their coverage at 90% or higher. These testers were able to increase coverage to 70% to 80%. Increasing coverage past this point proved to be difficult due to such factors as checks for “impossible” error conditions, hooks for future releases, and code that can be tested only with special hardware.

In order to model fault detection and removal, the researchers made some assumptions:

- Faults are not distributed evenly across the software. Instead, they tend to cluster in certain modules.
- The test team will have knowledge on which modules are more fault prone and will test these modules first.
- When a statement is exercised in functional test, all faults in that statement are detected and removed.

Using these assumptions, the researchers collected data on the number of faults detected and various levels of coverage. Fitting a curve to these data points, they found that the relationship between fault detection and statement coverage was modeled by the following function:

$$m(l) = M \sum_{i=1}^l P(i)(1 - K^{-(n-i+1)}) \quad (2.1)$$

where i indicates each of the blocks traversed, $P(i)$ is the probability of a defect in block i , K is the “error detection constant”, n is the total number of blocks in the system, M is the total number of errors in the system, and $m(l)$ is the number of errors detected by covering l blocks. The usefulness of this model is thwarted by the mysterious error detection constant K . When the authors were approached for a clarification on this point, they stated that they realized that there was a parameter missing in the equation, they had no evidence on

the source of the variance, so they used the variable K and stated that it was a function of programmer skill.

Inspecting this function, one can see that there is a direct relationship between fault detection and statement coverage. One can also see that at higher levels of coverage, this curve flattens out: that there is a clear point of diminishing marginal returns.

Based on this observation, the researchers concluded that increasing coverage is not cost-effective beyond 70% - 80%: that the benefit of additional fault detection does not outweigh the cost of additional coverage.

In 1993, Robert Grady published a report on his experience in measuring software quality at Hewlett-Packard [Grady, 1993]. Software quality at Hewlett-Packard includes functional testing, branch testing, reliability measures, and goals for the known defect density at release.

Charting the post-release defect reporting rate over time, and normalizing by the size of the software, Grady found that the reporting rate follows approximately a normal distribution over time. However, the products that did not meet internal certification had an amplitude of about three times that of the lowest-quality product which was certified. A strong relationship was found between the modules with high post-release defect density and those with high complexity: a point which supports the principle that to achieve high software quality, modules with high complexity should be rewritten.

While Grady measured the defect profile, he found that it varied widely across the organization. His observation was that since this data appears to be specific to each organization, software quality personnel are best off using internal data.

Most interestingly, he measured the branch coverage of products which had never had coverage measured before, and on which the testers felt that they were doing a thorough job. The actual level of coverage varied between 20% and 85%, with the average being about 55% over 22 products. This supports the finding at IBM that testers are unable to estimate their own coverage — that coverage measurement tools must be used.

One of the most relevant works for this study was published in 1992 by Sullivan and

Chillarege [Sullivan and Chillarege, 1992]. These researchers recognized that in order to achieve greater system reliability, it is important to gain an understanding of the faults which evade internal testing. So, they performed an empirical study on the field errors of three IBM software products — the MVS operating system, and the DB2 and IMS database systems.

To perform this work, the researchers studied over two hundred APARs on each product. An APAR is the fault report which is created by IBM when one of their products fails in the field. The information contained in each APAR includes a description of the error, the corrective action taken to fix the fault, and the severity. For each APAR, the researchers assigned the underlying fault to a fault class, assigned the defect type to a class, and analyzed the error trigger. The defect type is a broader category than the fault type — it seeks to contrast faults in broader categories. Finally, the error trigger describes the conditions under which the error will occur. Because all the faults analyzed are faults that evaded internal testing, it is clear that in order to reduce the number of errors in the field, the error triggers must be well-understood.

When the fault profile of the three products were compared with chi-squared tests, it became apparent that they represent three distinct profiles. This echoes Grady's finding, that fault profiles are so specific to applications and organizations that each organization should develop one internally.

By far, the most prevalent fault class in the APARs was *undefined state*. An undefined state fault occurs when the system enters a state which the developers had not anticipated and had not provided for. Undefined state faults represented 39.8% of all APARs on IMS, 20.27% of all APARs on DB2, and 16.77% of all APARs on MVS. For all three product, undefined state faults represented a large fraction of the high impact errors: 29% on IMS, 18% on DB2, and 25% on MVS. The remaining faults were scattered throughout the other fault classes, as shown in table 2.2. Pointer management was the next most common fault overall, but represented only 10.94% of the APARs in IMS, 10.36% in DB2, and 11.63% in MVS.

Since the undefined state faults were so prevalent, the researchers investigated their causes. There was little detail on their causes in MVS, but the researchers felt that in general, these faults related to concurrency. In IMS, approximately one third of the undefined state faults occurred when the program lost track of its current state, by setting a flag or a parameter incorrectly. An additional third of the undefined state faults represented missing cases in special case support. The rest were comprised mostly of incomplete or misunderstood specifications, or implementation faults. In DB2, nearly half of the undefined state faults related to missing cases, commonly involving boundary conditions. Others resulted from consistency checks being performed at the wrong time, frequently generating false alarms. In both database products, about two thirds of the undefined state faults resulted from omitted logic, and about one third resulted from incorrect logic.

The researchers concluded that the large population of undefined state faults makes improving the reliability of the products difficult, particularly IMS. One reason for this is that the same conditions which are not anticipated in the software development will not be anticipated in testing. Additionally, such reliability improvement measures as multiversion voting are likely to be unsuccessful because the same conditions are likely to not be anticipated in both versions. This echoes the findings of Shimeall and Leveson [Shimeall and Leveson, 1991], that multiversion programming and testing often fails in the same places: the problems that are difficult for the software developer are also difficult for the tester.

Table 2.2 summarizes the studies on major defect populations. In this table, you will notice major differences in the results of the different studies. This supports Grady's observation that the fault profile is a characteristic of each software project [Grady, 1993].

Study	Fault Classes	Population		
Ostrand & Weyuker	Data Definition	32%		
	Data Handling	22%		
	Decision and Processing	18%		
	Decision alone	18%		
	Other	10%		
Basili & Perricone	Interface	39.2%		
	Computation	19.4%		
	Control	16.2%		
	Data	14.0%		
	Initialization	11.2%		
Sullivan & Chillarege		DB2	IMS	MVS
	Undefined State	20.27%	39.8%	16.77%
	Allocation Management	8.1%	4.98%	6.58%
	Copying Overrun	5.4%	3.49%	2.38%
	Data	8.56%	4.48%	5.8%
	Interface	6.75%	7.47%	0%
	Memory Leak	3.6%	3.49%	0%
	Other	5.4%	3.99%	11.67%
	Pointer Management	10.36%	10.94%	11.63%
	Statement Logic	7.2%	8.46%	7.47%
	Synchronization	9.01%	4.48%	13.22%
	Unitinitialized Variable	6.3%	5.98%	7.51%
	Wrong Algorithm	9.01%	1.99%	0%
	PTF Compilation			8.09%
Unknown			8.93%	

Table 2.2: Populations of Major Fault Classes

3. Description of the Experiment

3.1 Description of the Software

The software studied is a large, mature, leading on-line transaction processing (OLTP) product. It contains over 5,000,000 lines of code and over 3000 modules. These modules vary widely in age, size, and even programming language. Because of the immensity of the full system, we chose to study a subset of its modules rather than the full system. There were two groups of modules used in this study, referred to as the older group and the newer group.

The older group is comprised of a sampling of modules from across the system. These modules vary in size, programming language and age. They have one common trait — they have had a high fault rate relative to other modules in the system. They are all from a version of the system which had been released approximately one year before the start of this study.

The newer group was chosen to contrast the older group. These modules are far newer modules, written less than three years ago, and are from a more recent release of the system. Rather than being from a wide assortment of functional areas, they are from one single area. Finally, they are produced by a software engineering team that has a strong reputation for excellence within the product development groups.

Though the age of the software is a major difference between the older and newer group, it is certainly not the only difference. Please bear this in mind while reading on.

3.2 Measuring Branch Coverage with EXMAP

The coverage information shown in this paper was produced with the EXMAP code coverage tool. EXMAP is an IBM-internal tool which measures statement and branch coverage. It provides reports which summarize the coverage of the selected modules, as shown in figure 3.1, or detailed information on the execution of each statement, as shown in figure 3.2.

PA	LOAD MOD	PROC	LISTING NAME	STATEMENTS:			BRANCHES:		
				TOTAL	EXEC	%	CPATH	TAKEN	%
1	LOADMOD1	TEST1	TEST1 LISTING *	45	43	90.9	34	19	55.8
2	LOADMOD1	TEST2	TEST2 LISTING *	21	21	100.0	15	13	86.7
3	LOADMOD1	TEST3	TEST3 LISTING *	10	2	20.0	4	1	25.0
4	LOADMOD1	TEST4	TEST4 LISTING *	103	77	86.7	42	34	80.1
Summary for all PAs:				179	143	79.9	95	67	70.5

Figure 3.1: Sample EXMAP Summary Report

The tests that were used in this study are contained in the “regression bucket”, the set of tests used in routine regression testing. These tests are all functionally-generated system tests. Along with the software, the regression bucket matures and changes over time. In order to get an accurate picture of how both groups had been tested prior to release, we used the version of the regression bucket that was used to test the release.

3.3 Collection of the Faults and Fault Class Assignment

Ninety-eight error reports were analyzed for this study. These errors had occurred in the field, and had all been analyzed and fixed at the time of the study. Each of these errors was reported in an internal error report called an APAR.

The information on all APARs is recorded in the RETAIN database. Each APAR starts with an error, typically reported by the customer, sometimes reported by IBM personnel performing alpha site testing. The error is diagnosed and a fault report is entered. This fault report is typically written by the engineer responsible for fixing the error, and typically contains a detailed description of the fix.

Using the information in RETAIN, each fault was analyzed and assigned to a fault class. The descriptions in RETAIN did not include a fault classification system: rather, they favored a textual description of the fault. We chose to use the following fault classification that has been used previously in studies of systems similar to the one discussed in this paper [Sullivan and Chillarege, 1992].

Allocation Management : One module deallocates a region of memory before it has completely finished using the region. After the region is reallocated, the original module continues to use it in its original capacity.

Copying Overrun : The program copies bytes past the end of a buffer.

Data Fault : An arithmetic miscalculation or other fault in the code makes it produce or read the wrong data.

Interface Fault : A module's interface is defined incorrectly or used incorrectly by a client.

Memory Leak : The program does not deallocate the memory it has allocated.

Pointer Management : A variable containing the address of data was corrupted. For example, a linked list is terminated by setting the last chain pointer to NIL when it should have been set to the head element in the list.

Statement Logic : Statements were executed in the wrong order or were omitted. For example, a routine returns too early under some circumstances. Forgetting to check a routine's return code is also a statement logic fault.

Synchronization : An error occurred in locking code or synchronization between threads of control.

Undefined State : The system goes into a state that the designers had not anticipated. For example, the program may have no code to handle an end-of-session message which arrives before the session is completely initialized.

Uninitialized Variable : A variable containing either a pointer or data is used before it is initialized.

Unknown : The fault report described the effects of the fault, but not adequately enough for us to classify it.

Wrong Algorithm : The program works, but uses the wrong algorithm to do the task at hand. Usually, these were performance-related problems.

Other We understood what the fault was, but could not fit it into a large enough category.

To classify the faults, the first source of information was the RETAIN database. RETAIN usually contained enough detail to assign the faults to fault classes, although sometimes it was necessary to examine the source code. When neither RETAIN nor the source code provided sufficient information to classify a fault, it was placed in the “Unknown” fault class. Two faults were classified as “Unknown”.

Ultimately, no faults were assigned to “Copying Overrun”, “Uninitialized Variable”, or “Other”. Therefore, we will not be discussing these fault classes later.

3.4 Relating Branch Coverage to Faults

When an APAR results in a software change, the modules affected are not modified directly. Instead, a patch file is created with the source code modifications to fix the fault. Lines added or modified are shown verbatim. When lines are deleted, it is actually replaced by a comment indicating for which APAR the line was deleted. When the patch file is patched into the source module, the proper location for each modification is determined using the location indices.

Location indices are similar to line numbers. Each line in the source file has a location index, and the location indices increase as you read down the file. Unlike line numbers, they are generated by hand. In other words, the programmer marks each line with a location index. The location index is treated as a comment by the compiler or assembler. In figure 3.2, the location indices are the numbers in the rightmost column. When the source file is first created, the delta between the location indices of two adjacent lines is fairly large. Then, when a line is added later, it is given a location index that is between the two adjacent location indices.

The location indices proved to be invaluable guides to determining whether or not a fault had been covered. Determining whether a portion of the fix had been covered was a simple matter of looking at the location index of the fixed line and seeing if that line or adjacent lines had been covered in testing. This coverage information came from the EXMAP annotated listings. In figure 3.2, the coverage information is shown in the column

4359:	STM	14, 12, 12(13)	8460500
4360:	LR	R15, =A(SAMPLE1)	8461000
4361:	LTR	R12, R12	8461500
4362>	BZ	LABEL1	8462000
4363 ◻	ST	R9, SAMPLE2	8463000
4364 ◻	L	R9, =A(SAMPLE3)	8464000
4365 ◻	L	R10, =A(SAMPLE4)	8465000
4366 ◻	CLI	FLAG1,SYMBOL1	8466000
4367 ◻	BNE	LABEL2	8467000

Figure 3.2: Excerpt from a Sample EXMAP Annotated Listing. The location indices are the numbers in the rightmost column. The coverage information symbol is shown immediately to the right of the line number.

: indicates that the line was executed.

◻ indicates that the line was not executed.

> indicates a logical expression that branched but did not fall through.

V indicates a logical expression that fell through but did not branch.

& indicates a logical expression which both fell through and branched.

These last two symbols are not shown in this figure, but are mentioned for completeness.

immediately to the right of the line number, and is described in the caption.

We assumed that the location of the fix was a good indicator of the location of the fault, and that the coverage of the patched locations was a good indicator of the coverage of the fault.

When determining whether or not the fault had been covered, we collected two numbers for each fault: the number of branches in the original code that were affected by the fix, and the number of these branches that were exercised during test. A branch is defined as being affected by a fix if any sequential statement on the branch is modified or if the conditional statement containing the branchpoint itself is modified. We refer to the branches affected in order to fix a fault as *affected branches*.

A branch is considered exercised if the statements on the branch are exercised. EXMAP provides codes to indicate if during execution, a conditional statement has branched, fallen through, both, or neither. Using these codes, it was possible to tell if a branch had been

exercised even when the branch itself contained no statements. For each fault, we collected data from the EXMAP output on how many of the affected branches had been exercised during testing. We refer to this quantity as *affected branch coverage*.

The raw data used in this study is included in appendix B. This includes the following information for each fault:

- the fault number,
- the number of the module which was modified because of the fault,
- the category of the fault,
- the number of branches in this module affected by the fix, and
- the percentage of the affected branches that were covered in regression testing.

For reasons of confidentiality, a fault number and module number are shown in place of the actual APAR number and module name.

4. Results

4.1 Analysis of the Faults Observed

Table 4.1 lists the types of faults analyzed in this study. The faults were selected at random from all faults on the modules mapped. The most prevalent fault class is interface faults, followed by undefined state and synchronization.

Figure 4.1 relates the number of faults to the number of affected branches. Notice the peak at size 1 and the exponential decay at sizes greater than 1. The average size of a fault in affected branches is 2.2.

Table 4.2 shows the average number of affected branches for each fault class. The number of affected branches relates to the complexity of fixing the fault by indicating the number of separate sections of code that must be touched by the fix. This is used as a measure of the cost of the fix [Wade, 1994]. As shown in this table, allocation management faults are by far the most complicated to fix. The fault class with the next highest number of affected branches is also one of the more common fault classes — synchronization. This data suggests that if software developers take extra care to prevent these faults, they will be rewarded with lower maintenance costs.

Fault Class	Number of Faults
Allocation Management	8
Data Fault	10
Interface Fault	20
Memory Leak	2
Pointer Management	2
Statement Logic Fault	5
Synchronization	17
Undefined State	17
Unknown	5
Wrong Algorithm	12

Table 4.1: Fault Class vs. Number of Faults

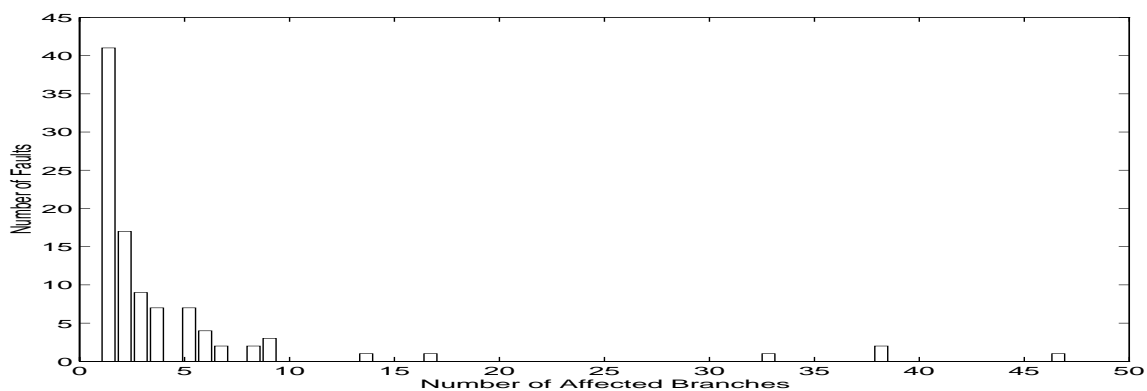


Figure 4.1: Number of Faults vs. Number of Affected Branches

Fault Class	Average Number of Affected Branches Per Fault
Allocation Management	11.9
Data Fault	2.3
Interface Fault	4.2
Memory Leak	1.0
Pointer Management	3.0
Statement Logic Fault	4.4
Synchronization	6.6
Undefined State	2.1
Unknown	4.8
Wrong Algorithm	2.8

Table 4.2: Average Number of Affected Branches by Fault Class

4.2 Affected Branch Coverage for All Classes of Faults

To determine how many of the faults are in covered code, we turn our attention to figure 4.2. The data shown in this table is the number of faults at various levels of affected branch coverage. This data is broken down between the older group and the newer group in table 4.3. A chi-squared test was used to determine if the data in table 4.3 represents different distributions for the older group and the newer group. The chi-squared probability that the data from the two groups is from two populations is 0.001. This implies that even though we have two separately collected sets of data, we should consider all the data as coming from one single source. Therefore, when we discuss how the affected branch coverage

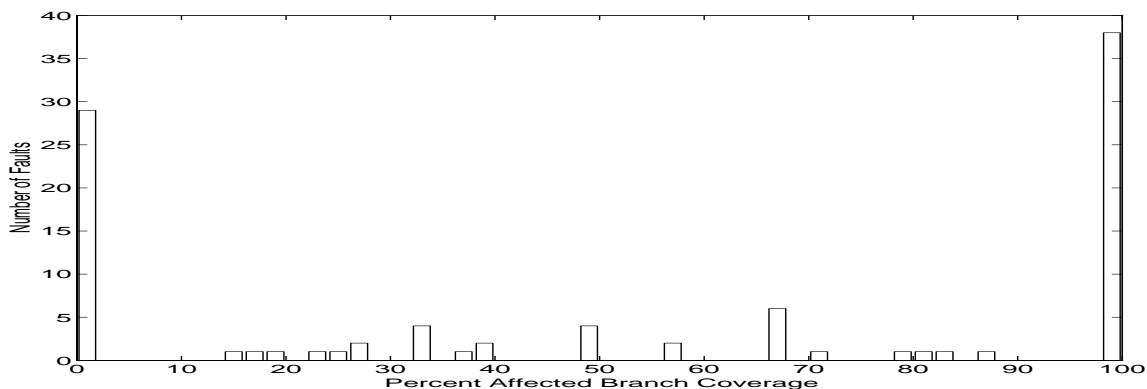


Figure 4.2: Number of Faults vs. Affected Branch Coverage

	0% - 25%	25% - 50%	50% - 75%	75% - 100%
Older Group	31	12	8	37
Newer Group	3	1	1	5
Overall	34	13	9	42

Table 4.3: Number of Faults vs. Overall Affected Branch Coverage

varies by fault type, we will not focus on which group the data came from.

Looking at this data, we see that about half of the faults occurred in covered branches. The overall affected branch coverage is 49.5% in the older group, 42.9% in the newer group, and 49.0% for both groups combined. In contrast, the branch coverage was 57.3% for the older group and 34.3% for the newer group.

Note the number of faults at the low end or the high end of the scale. A major factor behind this is the number of modifications that affect one branch only. Almost half of the faults studied affected only one branch, as shown in figure 4.1, and so have either 0% or 100% affected branch coverage.

Tables 4.4, 4.5, and 4.6 compare the affected branch coverage to fault class for the older group, newer group, and both groups respectively. A chi-squared test was used to measure the probability that the distribution of faults was different, for the fault classes that had a nonzero population in the newer group. The probability that the fault breakdowns are from different distributions was 0.108. This implies that even though we have two separately

	0% - 25%	25% - 50%	50% - 75%	75% - 100% i
Allocation Management	1	3	1	3
Data Fault	6	1	0	3
Interface Fault	6	2	2	8
Memory Leak	0	0	0	2
Pointer Management	2	0	0	0
Statement Logic Fault	1	0	1	2
Synchronization	3	3	1	4
Undefined State	4	1	1	11
Unknown	0	2	1	2
Wrong Algorithm	8	0	1	2

Table 4.4: Fault Class by Overall Affected Branch Coverage — Older Group

	0% - 25%	25% - 50%	50% - 75%	75% - 100%
Interface Fault	1	0	0	1
Statement Logic Fault	0	1	0	0
Synchronization	2	0	1	3
Wrong Algorithm	0	0	0	1

Table 4.5: Fault Class by Overall Affected Branch Coverage — Newer Group

	0% - 25%	25% - 50%	50% - 75%	75% - 100
Allocation Management	1	3	1	3
Data Fault	6	1	0	3
Interface Fault	7	2	2	9
Memory Leak	0	0	0	2
Pointer Management	2	0	0	0
Statement Logic Fault	1	1	1	2
Synchronization	5	3	2	7
Undefined State	4	1	1	11
Unknown	0	2	1	2
Wrong Algorithm	8	0	1	3

Table 4.6: Fault Class by Overall Affected Branch Coverage — Both Groups

collected sets of data, we should consider all the data as coming from one single source. Therefore, when we discuss how the number of faults differs over various levels of affected branch coverage, we will ignore whether the data was originally from the newer group or from the older group.

In these tables, note that the coverage of affected branches seems strongly dependent on

Fault Class	Overall Affected Branch Coverage
Allocation Management	33.7
Data Fault	30.4
Interface Fault	70.2
Memory Leak	100
Pointer Management	0
Statement Logic Fault	65.2
Synchronization	45.1
Undefined State	65.7
Unknown	52.6
Wrong Algorithm	40.6

Table 4.7: Overall Coverage of Affected Branches by Fault Class

fault class. For example, more than half of the data faults had an affected branch coverage of 25% or less, while almost two thirds of the undefined state faults had an affected branch coverage of 75% or more. This is confirmed in table 4.7, which contains the overall affected branch coverage by fault class for both groups of modules.

Pointer management, allocation management, data faults and wrong algorithm all have low affected branch coverage. For pointer management, there are not enough faults to demonstrate a trend. For wrong algorithm, the explanation is easy — many of these faults can be viewed as a documentation change. A common instance of a wrong algorithm fault is an inconsistency between the documented conditions under which an error message would appear and the actual conditions under which it did appear. This is the sort of fault which is often detected through usage. This software has been in the field for years. In a sense, this means that it has had years of testing done by the customers. A study comparing structural coverage in operational usage and functional testing found a high correlation: the sections of code that functional testers execute are likely to be the same sections of code that users execute [Ramsey and Basili, 1985]. It might be that wrong algorithm faults have a low affected branch coverage because the faults are associated with code *not executed* by the users, and covered code is also code *executed* by the users. In other words, the faults remain in the sections of code that do not get tested.

To explain the low affected branch coverage of allocation management and data faults,

we turn to the architecture of the software system itself. The software features a large amount of internal consistency checks. These internal consistency checks are similar to assertions in the C language. A violation results in an Abnormal End, otherwise known as an “Abend”. An abend is similar to an assertion in that it informs the users quite visibly that something has gone wrong and pinpoints the location at which the inconsistency was detected. Many abends are related to data inconsistencies. The explanation for the low affected branch coverage of data faults is that the abend system is so effective at detecting data faults that the remaining data faults are in the more obscure branches. In other words, the faults are detected by the tests.

The classes of faults for which there is an unusually high affected branch coverage are interface, memory leak, statement logic, and undefined state faults. There are so few memory leak faults that we cannot make a strong statement about them here. For the others, we turn to the fixes themselves. The typical fix for an interface, statement logic, or undefined state fault is to add branches to support a special case. Thus, the special case was not included in testing, while the average case executed the affected branches.

The major fault class not discussed yet is synchronization. The overall branch coverage of these faults is slightly under 50%. This data does not show a strong relationship between synchronization faults and affected branch coverage. This is not surprising given the faults themselves. These were typically faults that involved a small window in which race conditions could occur, and code coverage cannot tell us whether or not these conditions occurred in testing.

4.3 Contrasts Between Affected Branch Coverage on the Two Groups

The older group has an overall affected branch coverage of 49.5% when tested at a 57.3% rate of coverage. So, the older group has a somewhat greater density of faults in uncovered code than in covered code. The newer group has an overall affected branch coverage of 42.3% when tested at a 34.3% rate of coverage. So, the newer group has a somewhat greater density of faults in covered code than in uncovered code.

This finding may be a direct function of the greater maturity of the older group — in the older group, a greater portion of the code has been in existence for a greater amount of time. As stated earlier, the code has had billions of hours of testing by the users. In particular, the main sections of the code have been thoroughly tested over the years, and during that time faults in the main sections of the code have been detected and removed. In contrast, the newer group is still experiencing that maturing process.

Additionally, as the older software has aged, fewer of the original authors are still available. The people maintaining the code may not be familiar with all of its intricacies, and may miss an obscure branch in the course of a modification.

A pronounced difference can be seen in the class of faults, as shown in tables 4.4 and 4.5. The newer group shows a greater percentage of synchronization faults, though this is probably more closely related to the functionality of the newer group than to the fact that the software is newer. Yet the older group does show a greater incidence of undefined state, data, and interface faults. This may be because of the methodical engineering for which the newer group is known, involving a high level of teamwork and communication within the group.

5. Conclusions

This study characterized the relationship between fault classes and branch coverage, studying ninety eight different faults on a leading industrial on-line transaction processing system. The faults were analyzed to determine their class, the number of affected branches, and affected branch coverage.

We found that the more common fault classes were interface faults, data faults, and synchronization faults. Synchronization faults also appear to be among the more complex faults to fix based on the number of affected branches. Allocation management faults were by far the most complex. By taking extra pains to guard against these classes of faults, the software team can keep their maintenance effort in check. Fortunately, most faults affected only one or two branches.

We discovered that the coverage of affected branches varied significantly by the class of fault. For instance, data faults and wrong algorithm faults were far less likely to have been covered in testing. The affected branch coverage of wrong algorithm faults might be low because this type of fault is commonly found by the user. Prior studies have shown that code executed under functional test is usually also executed by the user. The remaining wrong algorithm faults are in code not often executed by the user, and not executed under functional test. In the case of data faults, the software is effective at watching for them and causing an ABEND when they occur. If coverage was increased, more of the data faults would probably be detected.

However, all of the fault classes with low affected branch coverage comprise only about one third of the total faults.

Undefined state, statement logic, and interface faults were typically in covered branches. Their fixes often involve adding special case branches. The explanation for the high affected branch coverage of these faults is that the affected branches have been executed by the average case, while internal testing has not included the special case. These fault classes represent about two-thirds of the total.

We found that overall, the software had an affected branch coverage of approximately 50%, indicating that many of the faults were in code that was covered in testing. This suggests that increasing branch coverage would offer limited gains in additional fault detection.

Our data suggests a greater density of faults on covered code in the newer group than in the older group. There are two explanations for this. First, the low affected branch coverage in the older group is in part a direct result of its maturity. Over the years, the software has been used actively, and the faults in the more common branches have been detected and removed. Second, there is a greater probability that the authors of the newer group are still available, while the older group may be maintained by someone unfamiliar with the software. Of the two people, the maintainer of the older group has a greater chance of missing a branch in the course of a large programming change.

This data suggests that increasing branch coverage is an effective way to increase detection of certain classes of faults. But to increase overall fault detection, it is more important to broaden the manner in which the code already covered is tested, and to try to introduce more special cases to the testing. Instead of looking at what *branches* have not been executed, look at what *functionality* related to these branches have not been executed. The gain of such analysis may be a small increase in branch coverage — but a larger increase in the variety of scenarios exercised in testing.

5.1 Areas for Further Study

There are many questions left to be answered on why various forms of structural testing are more effective at finding certain classes of faults and less effective at finding others. Until we answer these questions, we do not understand the benefits and limitations of structural testing.

Perhaps a good fault taxonomy has not yet been defined, resulting in certain classes of faults being grouped together erroneously. For instance, we found that many of our interface faults related to special cases which were not supported, yet there were a few

interface faults that were clear-cut inconsistencies. A better taxonomy might divide these two types of interface faults into two groups.

Perhaps a good taxonomy would involve the cause of the fault rather than its description. For instance, suppose we looked at faults caused by the programmer working from design specifications that did not include enough detail. If we learned that most testing methods were not effective at finding these faults, preventing these faults would assume a greater importance. In addition, the more we understand about the cause of a type of fault, the better we can become preventing it in the best case and testing for it in the worst case.

One possible explanation as to why such a study has not been performed is that determining the cause of a fault is difficult. The best approach is usually to consult with the software developer and see what was intended when the code was written. For instance, if a fault relates to an area in which the written specification was not complete, it is difficult whether or not the relevant requirement was incomplete: there could be clearly-communicated assumptions that fill in many gaps in written specifications. It is usually not possible for an outsider to navigate through the myriad of documents relating to a software project without some assistance from someone intimate with the project. The study proposed here might not be possible on anything but a recent project.

On a different note, it is very interesting that code executed under functional test is probably also executed by the users [Ramsey and Basili, 1985]. Sadly, that particular finding came from studying a small software product. It would be very useful to see if the finding holds for a very large software product such as the one studied here.

Appendix A. Glossary

This glossary contains definitions for the software engineering terms used in this paper. These definitions are from the IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1991]

assertion : A logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution. Types include input assertion, loop assertion, output assertion.

boundary value : A data that corresponds to a minimum or maximum input, internal, or output value specified for a system or component.

branch : A computer program construct in which two or more alternate sets of program statements is selected for execution.

branchpoint : A point in a computer program at which one of two or more alternative sets of program statements is selected for execution.

branch testing : Testing designed to execute each outcome of each decision point in a computer program.

code : In software engineering, computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler, or other translator.

code review : A meeting at which software code is presented to project personnel, managers, users, customers, or other interested parties for comment or approval.

data flow : The sequence in which data transfer, use, and transformation are performed during the execution of a computer program.

error : The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

failure : The inability of a system or component to perform its required functions within specified performance requirements.

fault : An incorrect step, process, or data definition in a computer program.

functional testing : Testing that ignores the internal mechanisms of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

inspection : A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include code inspection, design inspection.

mutation testing : A testing methodology in which two or more program mutations are executed using the same test cases to evaluate the ability of the test cases to detect the differences in the mutations.

path : A sequence of instructions that may be performed in the execution of a computer program.

path testing : Testing designed to execute all or selected paths through a computer program.

program mutation : A computer program that has been purposely altered from the intended version to evaluate the ability of test cases to detect the alteration.

regression testing : Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or components still complies with its specified requirements.

statement : In a programming language, a meaningful expression that defines data, specifies program actions, or directs the assembler or compiler.

statement testing : Testing designed to execute each statement of a computer program.

stepwise refinement : A software development technique in which data and processing steps are defined broadly at first and then further defined with increasing detail.

structural testing : Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing.

system testing : Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

test, test case : A set of test inputs, executions conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

test coverage : The degree to which a given test or set of tests addresses all specified requirements for a given system or component.

test incident report : A document that describes an event that occurred during testing which requires further investigation.

test procedure : Detailed instructions for the set-up, execution, and evaluation of results for a given test case.

testability : The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

testing : The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

unit : A separately testable element specified in the design of a computer software component.

unit testing : Testing of individual hardware or software units or groups of related units.

Appendix B. Raw Data

Table B.1 contains the raw coverage data for the older group. This table contains the following information:

- the fault number
- the module which was modified because of the fault.
- the category of the fault
- the number of branches in this module affected by the fix
- the percentage of the affected branches that were covered in regression testing

For reasons of confidentiality, a fault number and module number are shown in place of the actual APAR number and module name.

Table B.2 contains the raw coverage data for the newer group. The same fields are contained in table B.2 as in table B.1.

Fault	Type	Module	# Branches	% Covered
1	Allocation Management	16	3	33
2	Allocation Management	9	1	100
3	Allocation Management	10	38	13
3	Allocation Management	10	38	13
4	Allocation Management	15	1	58
5	Allocation Management	5	2	100
6	Allocation Management	10	3	67
7	Allocation Management	16	5	100
8	Data Fault	12	7	14
9	Data Fault	11	1	100
10	Data Fault	4	1	0
11	Data Fault	12	3	0
12	Data Fault	24	1	100
13	Data Fault	28	4	50
14	Data Fault	29	2	0
15	Data Fault	11	1	0
16	Data Fault	4	2	100
17	Data Fault	11	1	0
18	Interface Fault	4	6	100
19	Interface Fault	7	1	0
20	Interface Fault	14	6	83
21	Interface Fault	16	1	100
22	Interface Fault	11	5	40
23	Interface Fault	23	1	100
24	Interface Fault	18	1	100
25	Interface Fault	25	1	0
25	Interface Fault	30	1	0
26	Interface Fault	3	7	71
27	Interface Fault	3	3	67
28	Interface Fault	5	4	25
29	Interface Fault	6	33	81
30	Interface Fault	27	4	50
31	Interface Fault	31	1	0
32	Interface Fault	7	1	0
33	Interface Fault	14	5	80
34	Interface Fault	12	1	100
35	Interface Fault	4	1	0
36	Memory Leak	14	1	100
37	Memory Leak	27	1	100

Table B.1: Raw Coverage Results for the Older Group

fault	Type	Module	# Branches	% Covered
38	Pointer Management	8	1	0
39	Pointer Management	3	5	0
40	Statement Logic	13	1	100
41	Statement Logic	4	2	0
42	Statement Logic	4	6	67
43	Statement Logic	14	8	48
44	Statement Logic	16	1	100
45	Synchronization	4	47	57
46	Synchronization	17	3	33
47	Synchronization	14	9	0
48	Synchronization	20	2	100
49	Synchronization	21	2	100
50	Synchronization	11	1	0
51	Synchronization	18	9	22
52	Synchronization	18	1	100
53	Synchronization	20	3	33
54	Synchronization	3	7	14
55	Synchronization	8	1	1
56	Synchronization	13	8	38
57	Undefined State	3	1	100
58	Undefined State	12	2	100
59	Undefined State	18	1	100
60	Undefined State	19	2	0
61	Undefined State	5	1	100
62	Undefined State	5	3	67
63	Undefined State	25	1	100
64	Undefined State	26	1	100
65	Undefined State	29	1	0
66	Undefined State	22	4	0
67	Undefined State	4	5	100
68	Undefined State	4	2	100
69	Undefined State	16	3	0
70	Undefined State	27	1	100
71	Undefined State	7	4	100
72	Undefined State	17	2	50
73	Undefined State	27	1	100

Table B.1: Raw Coverage Results for the Older Group — page 2 (cont)

Fault	Type	Module	# Branches	% Covered
74	Unknown	1	1	100
75	Unknown	22	6	67
76	Unknown	17	9	33
77	Unknown	5	2	50
78	Unknown	7	2	100
79	Wrong Algorithm	21	14	57
80	Wrong Algorithm	3	2	0
81	Wrong Algorithm	16	1	0
82	Wrong Algorithm	32	2	0
83	Wrong Algorithm	3	2	100
84	Wrong Algorithm	21	1	0
85	Wrong Algorithm	14	2	0
86	Wrong Algorithm	3	2	0
87	Wrong Algorithm	4	1	0
88	Wrong Algorithm	8	4	0

Table B.1: Raw Coverage Results for the Older Group — page 3 (cont)

Fault	Type	Module	# Branches	% Covered
89	Interface Fault	44	1	100
90	Interface Fault	37	1	0
91	Statement Logic	40	5	40
92	Synchronization	34	4	100
93	Synchronization	35	5	0
93	Synchronization	36	12	25
94	Synchronization	38	1	100
95	Synchronization	39	1	100
96	Synchronization	41	1	0
97	Synchronization	42	3	67
98	Wrong Algorithm	43	1	100

Table B.2: Raw Coverage Results for the Newer Group

References

- [Basili and Perricone, 1984] V. Basili and B. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.
- [Basili and Selby, 1987] V. Basili and R. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions of Software Engineering*, SE-13(12):1278–1296, December 1987.
- [Beizer, 1990] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
- [Gelperin and Hetzel, 1988] D. Gelperin and B. Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, June 1988.
- [Girgis and Woodward, 1986] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Workshop on Software Testing*, volume 36, pages 64–73, July 1986.
- [Grady, 1993] R. Grady. Practical results from measuring software quality. *Communications of the ACM*, 36(11):62–68, November 1993.
- [Hennessy and Patterson, 1990] J. Hennessy and D. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufman Publishers, 1990.
- [Howden, 1982] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.
- [IEEE, 1991] IEEE, editor. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Inc., February 1991.
- [McCabe, 1976] T. McCabe. A complexity measure. *IEEE Transactions of Software Engineering*, SE-2:308–320, December 1976.
- [Ostrand and Weyuker, 1984] T. Ostrand and E. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289–300, 1984.
- [Piwowarski *et al.*, 1993] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *15th International Conference on Software Engineering*, pages 287–301. IEEE, April 1993.
- [Pressman, 1992] R. Pressman. *Software engineering : a practitioner's approach*. McGraw-Hill, 1992.
- [Putnam and Myers, 1992] L. Putnam and W. Myers. *Measures for excellence*. Yourdon Press, 1992.
- [Ramsey and Basili, 1985] J. Ramsey and V. Basili. Analyzing the test process using structural coverage. In *8th International Conference on Software Engineering*, pages 306–312. IEEE, April 1985.
- [Selby, 1986] R. Selby. Combining software testing strategies: an empirical evaluation. *IEEE Workshops on Software Testing*, 36(11):82–90, July 1986.
- [Shimeall and Leveson, 1991] T. Shimeall and N. Leveson. An empirical comparison of software fault tolerance and fault elimination. *IEEE Transactions of Software Engineering*, SE-17(2):173–182, February 1991.

- [Sims, 1990] C. Sims. Disruption of phone service is laid to a computer problem. *The New York Times*, 139:A1, column 5, January 1990.
- [Su and Ritter, 1991] J. Su and P. Ritter. Experience in testing the motif interface. *IEEE Software*, 8(2):26–33, March 1991.
- [Sullivan and Chillarege, 1992] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *22nd International Symposium on Fault Tolerant Computing*, volume 36, pages 475–484. IEEE, July 1992.
- [Wade, 1994] B. Wade, 1994. Personal communications with Barbara Wade of IBM, Santa Teresa Labs.