An Empirical Study of the Branch Coverage of Different Fault Classes

*[*]

Melissa S. Cline
Linda. L. Werner

UCSC-CRL-94-30
September 5, 1994

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

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 paper 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%, suggesting that increasing branch coverage would offer limited gains in fault detection.

Keywords: branch testing, code coverage, fault classification

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

Contents

1. Introduction

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. 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].

So therefore, 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 domain faults than computation faults, 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 commission than omission, 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.

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 2.3

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 examine the comparative effects of the age of the software to the fault class and affected branch coverage. We have found that the affected branch coverage is higher than the branch coverage in both older and newer software, though the difference is more vivid in the newer software.

2. Description of the Experiment

2.1 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.

2.2 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 2.1, or detailed information on the execution of each statement, as shown in figure 2.2.

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.

```
                                   STATEMENTS:      BRANCHES:
PA LOAD MOD  PROC   LISTING NAME   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 2.1: Sample EXMAP Summary Report

2.3 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.

```
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 2.2: 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.

2.4 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 2.2, 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 2.2, the coverage information is shown in the column 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 A. 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.

3. Results

3.1 Analysis of the Faults Observed

Table 3.1 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 3.1 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 3.2 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.

3.2 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 3.2. 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 3.3. A chi-squared test was used to determine if the data in table 3.3 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 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 3.1, and so have either 0% or 100% affected branch coverage.

Fault Class	Number of Faults
Allocation Management	8
Data Fault	10
Interface Fault	20
Memory Leak	2
Pointer Management	2
Statement Logic Fault	6
Synchronization	15
Undefined State	17
Unknown	4
Wrong Algorithm	12

Table 3.1: Fault Class vs. Number of Faults

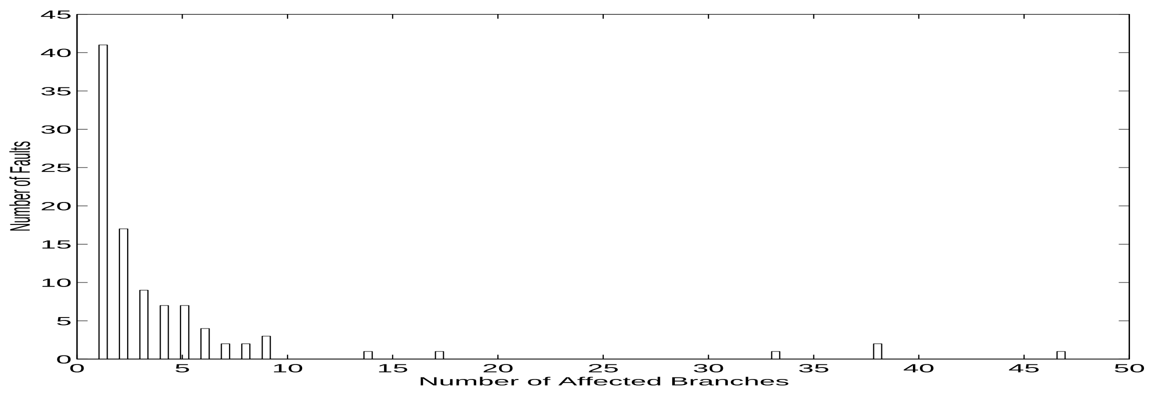

Figure 3.1: Number of Faults vs. Number of Affected Branches

Fault Class	Average Number of Affected Branches
Allocation Management	11.9
Data Fault	2.3
Interface Fault	4.2
Memory Leak	1.0
Pointer Management	3.0
Statement Logic Fault	3.8
Synchronization	6.6
Undefined State	2.1
Unknown	4.8
Wrong Algorithm	2.8

Table 3.2: Average Number of Affected Branches by Fault Class

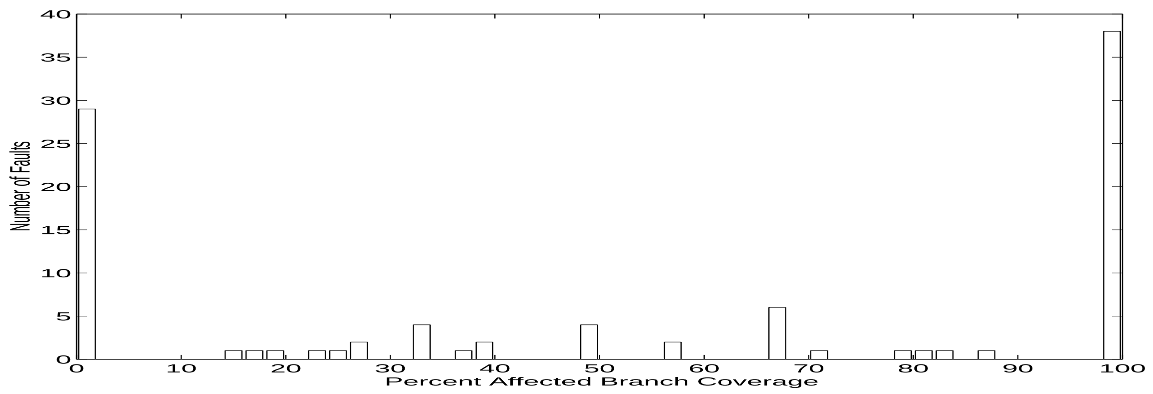

Figure 3.2: Number of Faults vs. Affected Branch Coverage

	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 3.3: Number of Faults vs. Overall Affected Branch Coverage

	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	3
Synchronization	3	3	1	4
Undefined State	4	1	1	11
Unknown	0	2	1	1
Wrong Algorithm	8	0	1	2

Table 3.4: 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 3.5: Fault Class by Overall Affected Branch Coverage — Newer Group

Tables 3.4, 3.5, and 3.6 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 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.

	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	3
Synchronization	5	3	2	7
Undefined State	4	1	1	11
Unknown	0	2	1	1
Wrong Algorithm	8	0	1	3

Table 3.6: Fault Class by Overall Affected Branch Coverage — Both Groups

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 3.7: Overall Coverage of Affected Branches by Fault Class

In these tables, note that the coverage of affected branches seems strongly dependent on 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 3.7, 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.

3.3 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 3.4 and 3.5. 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.

4. 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 immaturity. 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 class 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.

4.1 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.

References

[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.

[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.

[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.

[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.

[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.

Appendix A. Raw Data

Table A.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. There may be more than one module per 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

Table A.2 contains the raw coverage data for the newer group. The same fields are contained in table A.2 as in table A.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	4	100
4	Allocation Management	3	2	50
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	32	81
29	Interface Fault	7	1	100
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 A.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	2	50
43	Statement Logic	33	7	86
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	2	6	17
55	Synchronization	8	1	0
56	Synchronization	13	2	100
56	Synchronization	3	1	100
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 A.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 A.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 A.2: Raw Coverage Results for the Newer Group