GROUP BRANCH COVERAGE TESTING OF MULTI-VERSION SOFTWARE

Barbara Joan Swain

December 1986
CSD-860013

UNIVERSITY OF CALIFORNIA

Los Angeles

Group Branch Coverage Testing of Multi-Version Software

A thesis submitted in partial satisfaction of the

requirements for the degree of Master of Science

in Computer Science

by

Barbara Joan Swain

1987

The thesis of Barbara Joan Swain is approved.

[illegible]

David A. Rennels

[illegible]

Daniel M. Berry

[illegible]

John P. J. Kelly

[illegible]

Algirdas Avizienis, Committee Chair

University of California, Los Angeles

1987

ABSTRACT OF THE THESIS

Group Branch Coverage Testing of Multi-Version Software

by

Barbara Joan Swain

Master of Science in Computer Science

University of California, Los Angeles, 1987

Professor Algirdas Avizienis, Chair

Multi-version software is an approach to software fault tolerance that uses several redundant, but independently developed, versions of the software to mask errors in individual versions. Group branch coverage testing is a variant of branch coverage testing designed for testing multi-version software. This thesis introduces group branch coverage testing and offers an empirical evaluation of its effectiveness based on nineteen independently developed software versions from the Second Generation Experiment.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Branch Coverage Testing 2

1.2 Example of Branch Coverage Testing 7

1.3 Multi-Version Software 16

1.4 The Second Generation Experiment in Multi-Version Software 19

1.5 Group Branch Coverage Testing 21

1.6 Procedural Details 24

2 Analysis 29

2.1 Unreachable Code 29

2.2 Illegal Branches 32

2.3 Coverage Sets 33

2.4 Independence of Testing 34

2.5 Evaluation of Group Branch Coverage Testing 37

2.5.1 Comparison with Ordinary Branch Coverage Testing 37

2.5.2 Comparison with a New Acceptance Test 39

2.6 Similar Errors 40

2.7 A Detailed Example 44

3 Conclusions 48

References 50

Appendix A 52

Appendix B 54

LIST OF FIGURES

Page

Producing an Instrumented Program 2

Using an Instrumented Program 2

Classification of Branches 4

Sample Program 8

Instrumented Program 12

Coverage Report 14

Multi-Version Software System 16

Redundant Strapped Down Inertial Measurement Unit 20

Display Panel 22

Percentage of Branches Covered by Initial Acceptance Test 25

Percent Unreachable and Illegal Branches 30

Covered Set versus Uncovered Branches 34

Number of Testing Iterations per Version 35

Number of Faults Found 38

Number of Faults Found in UCLA Versions 39

Occurence of Faults 42

Occurence of Specification Related Similar Faults 44

Occurence of Non-Specification Related Similar Faults 45

ACKNOWLEDGEMENTS

The research described in this thesis was partially supported by NASA grant NAG1–512. This research would not have been possible without the gift of the TCAT™ Coverage Reporter program from Software Research Associates.

I would like to thank my committe, Dr. Avivizienis, Dr. Kelly, Dr. Berry, and Dr. Rennels, for serving on my committee. They were all extremely helpful under tight time pressure.

A special thanks to Dr. Kelly for introducing me to this work and getting me started. Many thanks for helping me to clarify my thoughts.

A special thanks to Dr. Avizienis, director of the UCLA Dependable Computing and Fault–Tolerant Laboratory.

Thanks to the members of the fault tolerance research group, Rong-Tsung Lyu, Ann Tai, Kam-Sing Tso, Bradford Ulery, Karen Dorato, and Werner Schuetz, for their support, encouragement, and stimulating technical discussions.

TCAT is a trademark of Software Research Associates.

Many heartfelt thanks to my boyfriend, Bradford Ulery, for his love, suppport, steady guidance, and undying encouragement, not to mention editing and illustrations. I could not have done it without you.

ABSTRACT OF THE THESIS

Group Branch Coverage Testing of Multi-Version Software

by

Barbara Joan Swain

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Algirdas Avizienis, Chair

Multi-version software is an approach to software fault tolerance that uses several redundant, but independently developed, versions of the software to mask errors in individual versions. Group branch coverage testing is a variant of branch coverage testing designed for testing multi-version software. This thesis introduces group branch coverage testing and offers an empirical evaluation of its effectiveness based on nineteen independently developed software versions from the Second Generation Experiment.

CHAPTER 1

Introduction

Multi-version software (MVS) systems are gaining acceptance in safety critical applications, such as the aerospace industry [Hil85, Mar82], and the nuclear power industry [Bis85, Ram81, Vog85]. The multi-version approach to fault tolerant software systems requires several redundant, but independently developed *versions* of the software. The versions are run in parallel and a decision algorithm determines final results based on consensus. The multi-version approach does not eliminate the need for careful and extensive software testing, but it does offer the possibility of some new approaches. This thesis shows that Group Branch Coverage Testing, one of these new testing approaches, tests multiple versions more thoroughly than ordinary Branch Coverage Testing.

Branch coverage testing is a method of testing software that requires that each branch be followed at least once [Adr82, Pra83]. This method requires that test cases be developed to cover branches and run, until all branches are covered. Developing test cases that test a program well is difficult; branch coverage testing helps by indicating what conditions the next test case should trigger. Group branch coverage testing is a variant of branch coverage testing designed for testing multi-version software. This method requires that branch coverage testing be performed for each version. Then each version is tested on all of the test cases developed during branch coverage testing of all of the versions. If only one version explicitly tests for a special case, for example $x=0$, then a test case to test that special case is developed. When all versions are tested on all of the test cases, this test case is included and all versions are tested for the special case.

1.1 Branch Coverage Testing

In order to perform branch coverage testing on a program, one must instrument the program so that it records which branches are followed when it is executed (See Figure 1a). The source code is instrumented by another program. This *instrumenter* modifies each branch so that it writes to an external *trace file* that it was covered. Branches are those sequences of statements to which control can be directed depending upon the evaluation of a condition. For example, in an ordinary **if** statement, the **then** part is one branch and the **else** part is the other branch.

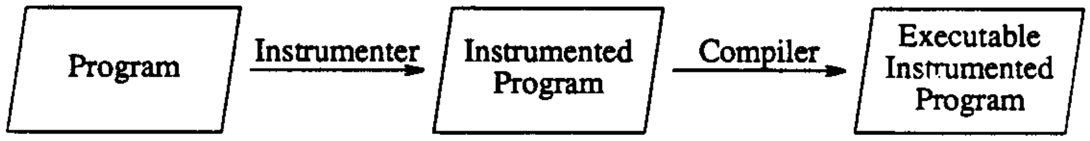

Fig. 1a. Producing an Instrumented Program.

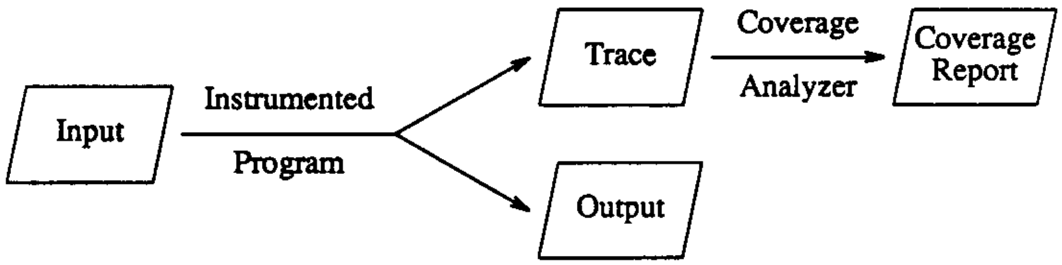

Fig. 1b. Using an Instrumented Program.

When the instrumented program is executed, it produces both the usual output that it produced before it was instrumented and a *trace file* that records which branches were executed (See Figure 1b). This trace file is analyzed by a *coverage analyzer* (the TCAT™, Test Coverage Analysis Tool, Coverage Reporter program was used for this research) to produce a *coverage report*, which summarizes which branches were and were not covered during the execution of the program. Instrumenting the program is almost automatic: after the program has been run through the instrumenter, a handful of statements have to be added to the instrumented version manually (See example in next section). The trace file is analyzed automatically by the coverage analyzer. Looking at the coverage report and instrumented code, and developing test cases is done entirely by humans.

The actual testing starts by running some test cases through the instrumented program. The choice of starting test cases is rather unimportant. The coverage report shows which branches have not been covered and have to be examined further. Each of these branches can be categorized as either unreachable, illegal, or coverable (See Figure 2).

An *unreachable* branch is a branch whose condition is necessarily false, regardless of the input. For example, the **then** branch is unreachable in the following code:

```
x ← abs(x)
if x < 0
    then $S_1$
    else $S_2$
```

The function abs always returns a value ≥ 0. Thus x is always non-negative, and the

TCAT is a trademark of Software Research Associates.

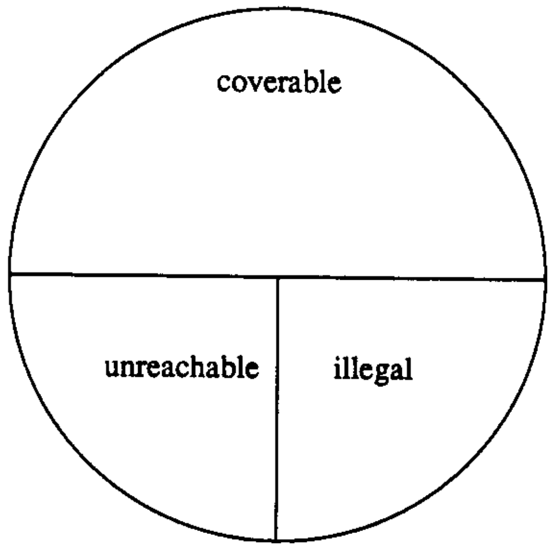

Fig 2. Classification of Branches.

condition x < 0 can never be true, regardless of the original value of x. An *illegal* branch is a branch that can not be covered during the execution of any valid input, but there is some *in*valid input whose execution causes the branch to be followed. *Valid* input is that which the specification states or implies may possibly occur. Stating how to respond to an input implies that the input may occur, even if the specification also states that the input will not occur. For example, if the specification specifies the proper response to a particular error condition, say a user giving a string that starts with a digit for his name, then that particular error condition is considered valid test input. *Invalid* input is that which the specification states or implies will not or cannot occur, and to which no response is specified. For example, if the specification states that an input value comes from a device that only generates values in a fixed range, e.g. a 16-bit channel, then testing that input with a value out of range is considered invalid. All other branches are *coverable*, i.e. they can be covered during the execution of some valid input.

Often illegal branches are the result of ambiguities or carelessness in the specification. The part of the specification that states that the conditions in the illegal branch will not occur deserves more evaluation. If it is decided that the specification is wrong in stating that the conditions will not happen, then the specification should be changed and the branch should become coverable.

For those branches which the coverage report indicates were not covered and examination shows to be coverable, test cases whose execution covers these branches are developed manually (See example in next section). The instrumented program is run on the new test cases, producing new trace files, from which a new cumulative coverage report is produced. This cycle is repeated until all coverable branches are covered, *completely covering* the program.

Branch coverage testing is similar to both statement coverage testing and path coverage testing. They are all forms of structural coverage testing, with each one requiring different structures, i.e. statements or branches or paths, to be covered [Adr82, Pra83]. Statement coverage testing requires that each statement be executed at least once. Statement coverage testing requires that a branch be covered only by requiring the statements of the branch to be covered. Path coverage testing requires that all paths be traversed. A path is one of the possible flows of control all of the way through a program. Path coverage testing requires that branches be covered in all possible combinations.

The major difference between branch coverage testing and statement coverage testing is that branch coverage testing requires more test cases. It is possible for a condition in a conditional statement to take on a value that corresponds to a branch that is not in the code. For example, it is possible for the condition in an **if** statement to be false even if the **else** branch is not written in the code. Branch coverage testing requires that these implied branches be covered; the instrumenter puts them in the code explicitly. Statement coverage testing ignores these implied branches. Branch coverage testing was designed as an improvement to statement coverage testing by forcing the conditions to take on all of their possible values.

The major difference between branch coverage testing and path coverage testing is that strict path coverage testing often requires an unbounded number of test cases, while branch coverage testing requires at most a bounded number of test cases; path coverage testing requires more test cases than branch coverage testing. If the number of iterations through a loop changes, then a different path has been taken through the program. For example, the following code has an infinite number of paths:

```
repeat
    read(x)
until x > 0
```

The input may have any number of negative numbers, causing any number of iterations through the loop. Even without an infinite number of paths through a loop, path coverage testing usually requires many more test cases than branch coverage testing. Consider code with three sequential **if** statements in it. With branch coverage testing, it is possible all of the branches will be covered after running two test cases, e.g. one causes the **then** branches to be covered and the other causes the **else** branches to be covered. However, path coverage testing requires that all possible combinations of truth values for the conditions be covered; there are 2 possible values for each condition or 2^3 paths. Path coverage testing rapidly becomes prohibitably expensive with the number of branches.

1.2 Example of Branch Coverage Testing

To help clarify the procedure of branch coverage testing, the beginning of this testing is shown for an example program using the tools used in this research. The example program determines the amount of the tax refund the user gets back or the amount of tax still owed. Obviously, this sample program is based on a hypothetical tax structure. The tax program:

```
program tax (input, output);
const
    taxbracket1 = 10000;
    taxbracket2 = 30000;
    taxbracket3 = 60000;
var
    income, withheld, tax, percentover : real;

procedure GetInput (var income, withheld : real);
begin
    repeat
        readln(income, withheld);
    until (income - withheld) >= 0;
end;

procedure ComputeTax (income, rate : real; var tax : real);
begin
    tax := 0;
    if rate > 0
        then begin
            tax := income * rate / 100;
        end;
end;

begin  { Tax }
    GetInput(income, withheld);

    if income < taxbracket1
        then ComputeTax (income, 10.0, tax)
    else if income < taxbracket2
        then ComputeTax (income, 20.0, tax)
    else if income < taxbracket3
        then ComputeTax (income, 30.0, tax)
    else ComputeTax (income, 40.0, tax);

    percentover := (withheld - tax) * 100 / income;

    if percentover > 0
        then writeln('Your tax refund is ', percentover:0:2,
                '% of your income')
        else writeln('You still owe ', -percentover:0:2,
                '% of your income in taxes');

end.
```

Fig. 3. Sample Program

Figure 4 shows the results of instrumenting this program. The instrumenter is language dependent and the one used in this example is for Pascal programs. Most of the instrumentation is done automatically. However, with this instrumenter, the instrumented program must be modified manually to include the parts in bold print in Figure 4. "#include" lines automatically include the named file at that place when the program is compiled. The file SRA.instr.i contains the procedures EntrMod, SegHit, and ExtMod. The files SRA.const.i, SRA.type.i, and SRA.var.i hold the constants, types and variables, respectively, that the code in SRA.instr.i needs. SRA stands for Software Research Associates who produced these branch coverage testing tools. The external file TRACE is the file to which the trace file is written.

The procedure EntrMod writes to the trace file that a particular procedure was entered. The procedure SegHit writes to the trace file that a particular branch was covered. The branch is identified by the parameter to SegHit and the procedure in which it is. For example, the branch that includes the call "SegHit(3)" is identified as branch 3 of the procedure. The procedure ExtMod writes to the trace file that a particular procedure was completed (exited).

```
program tax (input, output, TRACE);
const
     SRAmodvar = 'tax';
     taxbracket1 = 10000;
     taxbracket2 = 30000;
     taxbracket3 = 60000;
#include SRA.const.i
type
#include SRA.type.i
var
     income, withheld, tax, percentover : real;
#include SRA.var.i
#include SRA.instr.i
procedure GetInput (var income, withheld : real);
const
     SRAmodvar = 'GetInput';
begin
     EntrMod(2, 8, SRAmodvar);
     SegHit(1);

     repeat
          SegHit(2);
          readln(income, withheld);
     until (income - withheld) >= 0;

     ExtMod(SRAmodvar, 8)
end;

procedure ComputeTax (income, rate : real; var tax : real);
const
     SRAmodvar = 'ComputeTax';
begin
     EntrMod(3, 10, SRAmodvar);
     SegHit(1);

     tax := 0;
     if rate > 0
          then begin
               SegHit(2);
               begin
               tax := income * rate / 100;
               end
          end
          else SegHit(3);

     ExtMod(SRAmodvar, 10)
end;

begin  { Tax }
     rewrite(TRACE);
     EntrMod(9, 3, SRAmodvar);
     SegHit(1);
```

```
    GetInput(income, withheld);

    if income < taxbracket1
         then begin
              SegHit(2);
              ComputeTax (income, 10.0, tax)
          end
    else begin
         SegHit(3);
         if income < taxbracket2
              then begin
                   SegHit(4);
                   ComputeTax (income, 20.0, tax)
               end
    else begin
         SegHit(5);
         if income < taxbracket3
              then begin
                   SegHit(6);
                   ComputeTax (income, 30.0, tax)
               end
    else begin
         SegHit(7);
         ComputeTax (income, 40.0, tax)
    end;
    end;
    end;

    percentover := (withheld – tax) * 100 / income;

    if percentover > 0
         then begin
              SegHit(8);
              writeln('Your tax refund is ', percentover:0:2,
                   '% of your income')
          end
    else begin
         SegHit(9);
         writeln('You still owe ', –percentover:0:2,
                   '% of your income in taxes')
    end;

    ExtMod(SRAmodvar, 3)
end.
```

Fig. 4. Instrumented Program

The instrumented program is run on the test case "15000, 200". The program produces the following output:

```
You still owe 18.67% of your income in taxes
```

The program also produces a trace file, which is shown in Appendix A. Both the output and the trace file are particular to the test case; a different test case will not necessarily produce either the same output or the same trace file. Since the trace file is always written to the file named TRACE, TRACE has to be renamed before the next test case is run through the instrumented program. The instrumented program could be further modified to read in the name of the file where the trace file should be written.

The output is saved to be compared with correct results. The trace file is analyzed by the coverage analyzer. The coverage analyzer, TCAT Coverage Reporter, produces a coverage report (See Figure 5).

Page 1

```
TCAT Coverage Analyzer.  COVER Version 1.8 (80 Column)*
 (c) Copyright 1984 by Software Research Associates

Selected COVER System Option Settings:
(1 implies "on", 0 implies "off")


   Histogram Report       --    0
   Not Hit Report         --    1
   Cumulative Report      --    1
   Log Scale Histogram    --    0
   Banner Text:           --
   Past Report            --    0
   Sorted Module Names    --    1


   Options Read:   3
```

Page 2

```
TCAT Coverage Analyzer.  COVER Version 1.8 (80 Column)
 (c) Copyright 1984 by Software Research Associates

+------------------------------------------------------------+
|                 |     This Test        | Cumulative Summary |
+                 +----------------------+--------------------+
|                 |        No  Of        |        No  Of      |
|Module Number Of|No. Of  Segments  C1%|No. Of Segments  C1%|
|Name:  Segments:|Invokes  Hit    Cover|Invokes  Hit   Cover|
+------------------------------------------------------------+
|ComputeTax      3|   1       2    66.67|   1      2    66.67|
|GetInput        2|   1       2   100.00|   1      2   100.00|
|tax             9|   1       4    44.44|   1      4    44.44|
+------------------------------------------------------------+
|Totals         14|   3       8    57.14|   3      8    57.14|
+------------------------------------------------------------+

 Current test message (saved in archive):
```

*Output has been modified to fit on page.

Page 1

```
TCAT Coverage Analyzer.  COVER Version 1.8 (80 Column)*
(c) Copyright 1984 by Software Research Associates

Selected COVER System Option Settings:
(1 implies "on", 0 implies "off")


   Histogram Report     --   0
   Not Hit Report       --   1
   Cumulative Report    --   1
   Log Scale Histogram  --   0
   Banner Text:         --
   Past Report          --   0
   Sorted Module Names  --   1


   Options Read:   3
```

Page 2

```
TCAT Coverage Analyzer.  COVER Version 1.8 (80 Column)
(c) Copyright 1984 by Software Research Associates

+-----------------------------------------------------------+
|                |     This Test        | Cumulative Summary |
+                +---------------------+---------------------+
|                |       No  Of        |         No  Of      |
|Module Number Of|No. Of  Segments  C1%|No. Of Segments  C1% |
|Name:  Segments:|Invokes  Hit    Cover|Invokes  Hit    Cover|
+-----------------------------------------------------------+
|ComputeTax     3|   1      2     66.67|   1      2     66.67|
|GetInput       2|   1      2    100.00|   1      2    100.00|
|tax            9|   1      4     44.44|   1      4     44.44|
+-----------------------------------------------------------+
|Totals        14|   3      8     57.14|   3      8     57.14|
+-----------------------------------------------------------+

Current test message (saved in archive):
```

*Output has been modified to fit on page.

Page 3
```
TCAT Coverage Analyzer.  COVER Version 1.8 (80 Column)
(c) Copyright 1984 by Software Research Associates

C1 Not Hit Report.

No.    Module Name:          Segment Coverage Status:


   3: ComputeTax                3

   2: GetInput              All Segments Hit.  C1 = 100%

   1: tax                       2    5    6    7    8

Number of Segments Hit:         8
Total Number of Segments:      14
C1 Coverage Value:             57.14
```

Fig. 5. Coverage Report

The first page of the coverage report tells which options were chosen. In this example, it was requested that the output be sorted by procedure (module) name. For larger programs, this helps a tester find the information for a particular procedure. A "Not Hit Report" was also requested. The Not Hit Report shows which branches have not been covered. The branches that have not been covered have to be analyzed manually, in order to develop test cases whose execution will cover them.

The second page gives a summary. It shows in general how much was done with the current test case, "This Test", and with all of the testing so far, "Cumulative Summary". The column labeled "Number Of Segments" lists the number of branches in each procedure, plus one for entering the procedure. The columns labeled "No. Of Invokes" list how many times the procedure has been called for the current test case and cumulatively, respectively. The columns labeled "No Of Segments Hit" list how many of the branches in the procedure have been covered. The columns labeled "C1% Cover" list the percentage of branches covered in the procedure. For example, the first line of the table says that procedure "ComputeTax" has three branches, two or 66.67% of which were covered the one time it was called. The cumulative information is exactly the same, because there were not any previous test cases.

The third page has the Not Hit Report. This is the page from which the tester work most of his work. For each procedure, the branches which have *not* been covered are listed by number. This number corresponds to the number in the call to SegHit of that branch. If all of the branches have been covered, the message "All Segments Hit. C1 = 100%" appears instead of the list of uncovered branches.

The tester looks at the first row of the coverage report and notes that branch 3 has not been covered. She then looks at a copy of the instrumented program and notes that branch 3 is the **else** for the **if** statement with condition "rate > 0". The tester then looks for a way to make the parameter "rate" ≤ 0. Looking for all of the calls to "ComputeTax", the tester finds four calls in the main program. Each of the calls passes in a positive constant for the parameter "rate". There are no other calls to procedure "ComputeTax". The tester then marks branch 3 of "ComputeTax" as unreachable, and does not consider it further.

The tester looks at the next row of the coverage report and notes that procedure "GetInput" is already completely covered. She does not consider "GetInput further".

The tester then looks at the last row of the coverage report and notes that branches 2, 5, 6, 7, and 8 of program "tax" have not been covered yet. The tester then looks at the instrumented program and notes that if branch 6 is covered, then branch 5 is necessarily covered, because branch 6 is nested inside of branch 5. Likewise for branches 8 and 7. Thus, the tester only has to develop three test cases, instead of five.

The tester then looks more closely at branch 2 in program "tax". It is the **then** branch with condition "income < taxbracket1". After noting that taxbracket1 is a constant with value 10,000 and that income is an input variable, the tester develops the test case "5000, 100". Likewise, the tester develops test cases for branches 6 and 8.

Consider the test case "0, 0". There is no branch that forces a test case with "income" = 0. When the value of "percentover" is computed, the program aborts trying to divide 0 by 0. Branch coverage testing does not necessarily, or even likely, find all of the faults in a program. But it does require that a reasonably thorough effort be made in the testing.

1.3 Multi-Version Software

Multi-version software is an approach to software fault tolerance. It involves the independent development of several versions of the software. The versions are run in parallel and their results passed to a decision algorithm (See Figure 6).

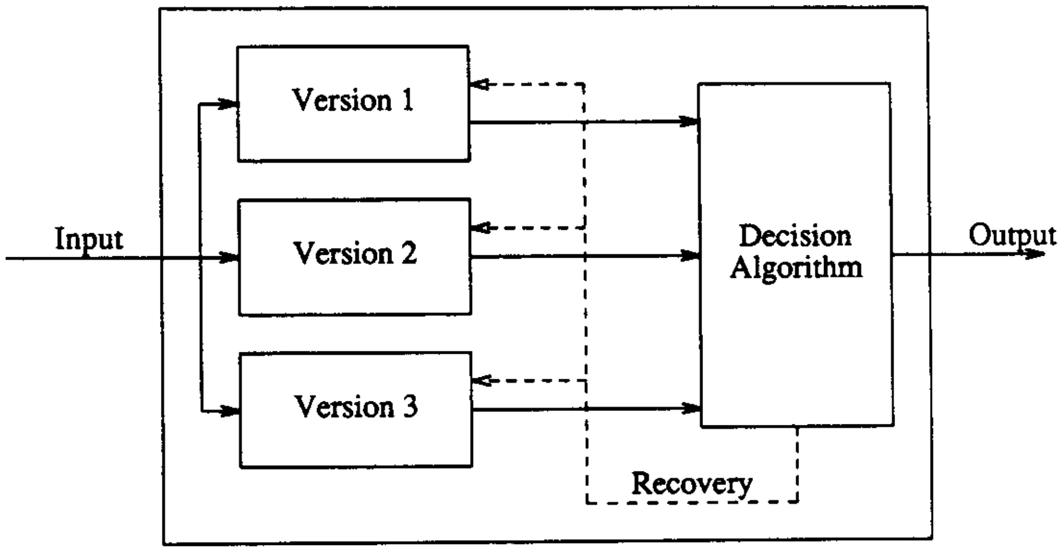

Fig. 6. Three-Version System.

The *decision algorithm* collects values from all of the versions and determines the final result for each variable based on consensus. Variables are passed through an external call to the decision algorithm, at *decision points*. The results of the decision algorithm are either provided as final outputs of the system, or intermediate variables, which may be used for recovery of faulty versions [Tso86]. The arrows from the decision algorithm to the versions show the consensus value(s) being returned to the versions for use in further computations. The only restrictions on decision points are the order in which they occur and the parameter passing protocol.

"Error" "failure" and "fault" are terms useful in describing software that does not satisfy requirements and discussing fault tolerance. An *error* is any incorrect state that occurs during program execution. Typically, errors are corrupted data. A *failure* is an error that manifests itself, i.e. any deviation from the acceptable service of the system. A *fault* is the identified or hypothesized cause of an error or failure. A *software fault* is the immediate cause of an error or failure that exists in the software. Often the underlying cause of a software fault is another fault, e.g. in the design [Avi84].

The basic premise of the multi-version approach is that errors in a version will be masked in the decision algorithm by the correct results of the other versions in the system. This premise does not depend on the independence of errors, but just a low probability of multiple versions having similar errors. *Similar results* are defined to be two or more results (good or erroneous) that are within the range of variation allowed by the decision algorithm [Avi84], i.e., two or more values for the same output variable on the same input that the decision algorithm treats as equivalent. *Coincident errors* are defined to be two or more errors that occur on the same input [Eck85]. Coincident errors do not necessarily appear equivalent to the decision algorithm, or even necessarily involve the same output variable. *Similar errors* are both coincident errors and similar results, i.e., two or more values for the same output variables on the same input that are erroneous and that the decision algorithm treats as equivalent.

The premise of low probability of similar errors does not reduce the need for testing. It increases the need for testing, because the premise is based, in part, on the assumption that the versions have been thoroughly tested. Consider an acceptance test for the versions of some MVS system that happens not to test a part of the input domain that rarely occurs. When the system eventually encounters input from that part of the domain, each of the versions is more likely to produce an error than when running on input from a well-tested part of the domain. Since errors are more likely to occur, it is also more likely that similar errors will occur. Thorough testing is crucial for MVS systems.

Multi-version software is used for applications that need extremely high reliability. The reliability of the system depends on the reliability of the versions that compose the system. The software must be thoroughly tested to ensure that it is of the highest possible quality. MVS systems require more than that each version produce a correct result. There may be more than one acceptable response to some input, and the decision algorithm must also be able to determine a correct result from the results of all versions. Thus, the versions must produce results that the decision algorithm recognizes as equivalent. Exceptions and special cases are more likely to have more than one acceptable result. But it is in these unusual cases that it is most important for the MVS system to work and to be able to obtain a consensus.

Vouk *et al.* suggest that functional testing may not come close to covering programs [Vou86]. Branch coverage testing should be done explicitly; complete branch coverage is unlikely to occur as a by-product of another testing strategy.

1.4 The Second Generation Experiment in Multi-Version Software

The NASA Langley Research Center has been sponsoring the Second Generation Experiment in multi-version software since 1984. The emphasis of this experiment is the evaluation of the reliability of MVS systems as compared to their components individually. This includes the modeling and analysis of MVS systems. Another goal of the experiment is further development of a multi-version programming methodology, particularly as related to design and testing. The University of California, Los Angeles, the University of Illinois at Urbana-Champaign, North Carolina State University, and the University of Virginia, as well as the National Aeronautics and Space Agency, Charles Rivers Analytics and Research Triangle Institute are involved in the experiment.

During the summer of 1985, 40 computer science graduate students from the four universities independently developed 20 Pascal program versions from a common specification. There were five teams of two programmers at each of the universities. Programmers were not permitted to discuss work-related issues with any member of another team. Electronic mail was used for all work-related communication; this communication was restricted to questions and answers between the programmers and a central project coordinator, who served as a specification arbitrator. Copies of each question and answer pair were sent to all teams. By the end of the summer, all of the versions had passed a preliminary acceptance test of 75 test cases.

A Redundant Strapped Down Inertial Measurement Unit (RSDIMU) is used in avionics to determine vehicle acceleration [CRA85]. An RSDIMU contains eight linear accelerometers on the four faces of a semioctahedron, a square pyramid whose faces are equilateral triangles. Each face has two sensors. Each of these sensors measures the component of acceleration along its axis. The software uses the redundant measurements of all of the working sensors to estimate the vehicle's acceleration. The sensors may fail before or during a flight. A face is considered to be failed if both of its sensors are failed. The sensors do not necessarily give perfect readings. This is modeled by considering the sensor's reading to be made up of the correct value and noise. A sensor may fail slowly with its readings becoming increasingly noisy.

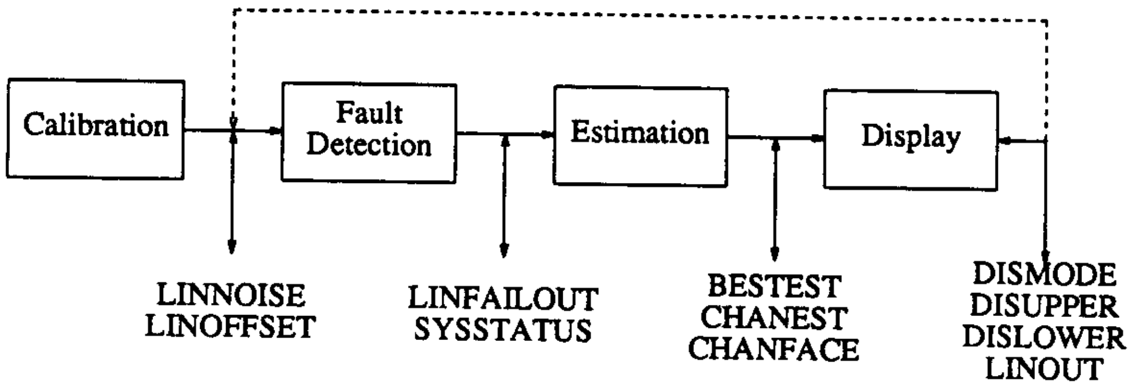

Fig. 7. Redundant Strapped Down Inertial Measurement Unit

Each version is organized into four modules (See Figure 7). The modules are defined by the decision points at the end of them.

The first module calibrates the sensors and identifies pre-flight sensor failures. A list of previously failed sensors is provided as input to the RSDIMU. Before the flight, a pre-determined number of sensor readings, taken when the vehicle is at rest, are recorded for each sensor to be used for calibration. A sensor is determined to have failed due to excessive noise, if the standard deviation of these calibration readings is too high. Sensor readings are converted from units of volts to meters/second^2.

The second module determines which, if any, additional sensors have failed, and reports all failed sensors in variable "LINFAILOUT". A sensor is determined to have failed in flight, if its reading is inconsistent with the readings of the other operational sensors. Two operational faces with all four sensors working are necessary to check the consistency of the readings of the sensors. Some versions did the consistency check by enumerating each combination of possible sensor failures. Other versions used a general algorithm to perform the consistency check. For some versions this part of the code had the most branches, because the different possibilities of sensor failures were handled separately. If it is not possible to do the consistency check, the system is declared non-operational as reported in variable "SYSSTATUS".

The third module estimates the vehicle's three-dimensional acceleration vector by finding the least squares fit to the readings from the up to eight operational sensors. Since this is the most critical computation in the RSDIMU, five redundant channels are provided for reporting it. The four additional channels carry estimates of the vehicle's acceleration based on the readings from the sensors on pairs of faces.

Various information can be displayed on a display panel (See Figure 9) driven by the RSDIMU. The display mode appears in the upper, left corner, in the mode indicator. The information appears in the two longer rows. The last module displays information on this display panel. There is a separate output variable driving each of the rows and the mode indicator. Each bit of the output is devoted to one segment of the display. This part of the code was not exercised by the preliminary acceptance test, and, not surprisingly, many faults were found in this part of the code.

1.5 Group Branch Coverage Testing

Although branch coverage testing requires that all of the conditions explicitly in the code be tested , it usually does not find a special case that is not handled properly. For example, consider the following code fragments that invert an input value and

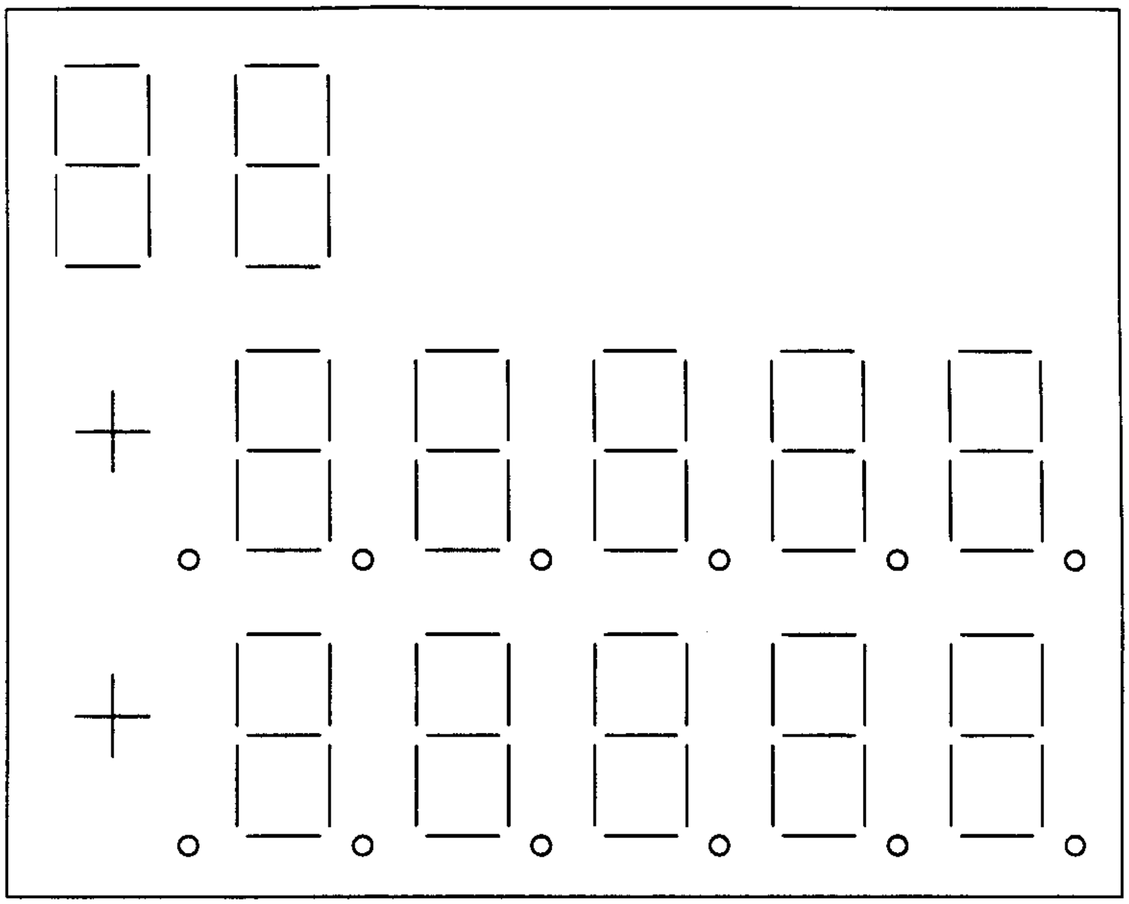

Fig. 8. Display Panel.

store the result in y:

Code A	Code B

read(x)
if x=0
    then write error message
    else y ← 1/x
write(y)
	read(x)
	
	y ← 1/x
	
	write(y)

It is unlikely that ordinary branch coverage testing would reveal the lack of a check on the value of "x" in B.

Group branch coverage testing is a variant of branch coverage testing designed for testing multiple versions. The versions are tested using branch coverage testing as usual. However, instead of using test cases only for testing the version for which they were developed, all versions are tested with all test cases. Because the versions were developed independently, their branches will not necessarily correspond exactly – even among versions that handle a special case correctly.

It is reasonable to expect that group branch coverage testing would be more effective than ordinary branch coverage testing. It takes only one version which explicitly handles a rare special case to reveal which, if any, other versions handle that special case incorrectly. A version that uses a loop structure to solve a problem is tested more thoroughly if there is a version that breaks down the problem into cases; the cases are likely to include extreme values and other cases that are more likely to have been overlooked in the loop structure. In the following example, both fragments are supposed to search linearly for one of four faces that is not OK and assign it to variable "badface":

Code A

```
if not OK[1]
    then badface ← 1
else if not OK[2]
    then badface ← 2
else if not OK[3]
    then badface ← 3
else if not OK[4]
    then badface ← 4
else badface ← −1
```

Code B

```
i ← 1
badface ← −1
while (badface < 0) and (i < 4)
    if not OK[i]
        then badface ← i
    else i ← i + 1
endwhile
```

It would be easy to cover code B without revealing that it does not check face 4, because the condition of the **while** loop should be "(badface < 0) and (i ≤ 4)".

However, the test case whose execution covers the last **then** in code A reveals the fault.

1.6 Procedural Details

Of the 20 versions that were produced for the Second Generation Experiment, one version, UIUC2, was not available for analysis. The remaining 19 versions are discussed in the rest of the paper.

When the 19 versions of this experiment were instrumented, several other changes were made to make testing easier. To the versions, the calls to the decision algorithm look like any other calls to a procedure. Since the decision algorithm is external to the versions, it was easy to replace it by a substitute decision algorithm that merely copies the values of its parameters to output. Each version was compiled with some extra code needed because of the instrumentation, the substitute decision algorithm and a harness.

There are no exact rules for the selection of test cases for group branch coverage testing. Seventy-five test cases were developed for the preliminary acceptance test in the Second Generation Experiment. For convenience, testing started with these 75 test cases.

Because the versions had passed the preliminary acceptance test, the output values for these test cases were assumed correct without examination. However, only some of the output variables were checked during the preliminary acceptance test. Some of the errors found by the later testing, which did check all the output variables, would have been found during the preliminary acceptance test, if all variables had been checked then. Many of the faults found through this analysis were not subtle, but still had not been caught by the preliminary acceptance test. The faults found in this analysis are skewed toward obvious faults that are usually found and corrected early. Table 1 shows how little of the code in the versions was tested by the preliminary acceptance test. The percentage of branches covered is not linear with the number of test cases needed. The execution of the first few test cases covers disproportionally many branches. The execution of the last few test cases typically covers only one additional branch per test case.

Version	Covered	Total	Percentage
UIUC1	179	251	71.31%
UIUC3	198	263	75.29%
UIUC4	213	306	69.61%
UIUC5	306	352	86.93%
NCSU1	353	465	75.91%
NCSU2	206	307	67.10%
NCSU3	269	493	54.56%
NCSU4	249	294	84.69%
NCSU5	307	406	75.62%
UVA1	390	530	73.58%
UVA2	335	423	79.20%
UVA3	403	816	49.39%
UVA4	259	316	81.96%
UVA5	430	531	80.98%
UCLA1	276	324	85.19%
UCLA2	233	329	70.82%
UCLA3	268	374	71.66%
UCLA4	303	418	72.49%
UCLA5	186	243	76.54%

Table 1. Percentage of Branches Covered by Initial Acceptance Test.

A cumulative coverage report listing all branches not covered in the preliminary acceptance test was produced for each version. The preliminary acceptance test provided a base of covered branches for each version that did not need further analysis. The process of developing test cases whose execution will cover particular branches must be individualized for each version, because it involves looking at the source code of each version by hand. The manual analysis of the versions often revealed that the execution of a test case that had already been developed would cover a branch in a new version. When this occurred, the test data were reused. It took several iterations of examining uncovered branches, developing test data and running the new test cases before the versions were completely covered.

A natural way to perform group branch coverage involves covering the versions sequentially. First, one version is covered. Next, another version is chosen and it is tested on all of the test cases that the first version needed. Any additional test cases necessary to cover all of the branches in the second version are developed. *Both* versions are then tested with these new test cases. This cycle of testing a new version on all existing test cases, developing new test cases, and then running the previously covered versions on the new test cases is repeated until all versions are covered. The order in which versions are selected for testing is unimportant, because each version is completely covered and each version is tested on each test case. It is possible that some of the versions that are tested later will be completely covered by the test cases that have already been developed, and thus will not need any new test cases.

Because of the large number of versions that needed to be tested, this process would have been too time consuming. Instead, in this analysis, several versions were tested in parallel, but each version was still tested separately. An attempt was made to re-use test cases that had already been developed; after the conditions necessary for covering a branch were well understood, then available test cases were searched to see if one of them could be used to cover the branch. This method may have required developing extra cases, because it was sometimes difficult to recognize when the execution of a previously developed test case would cover a branch in a new version.

The increase in the speed with which testing was accomplished, made up for the cost of the extra test cases in this case.

The purpose of branch coverage testing is to develop a *coverage set* for the program being tested, i.e., a collection of test cases that, when executed, covers the program completely. The coverage set that was found for each version during this testing will be referred to as *the* coverage set for that version. The union of all 19 versions' coverage sets obviously covers all of the versions. Group branch coverage testing requires that all versions be tested on this union of the coverage sets. Thus, group branch coverage testing makes use of information in all versions to test all versions better.

When comparing the results from the versions, it is reasonable to expect integer-valued and boolean-valued results to agree exactly. However, correct real-valued results typically do not agree exactly. For purposes of determining correct results, real-valued results are grouped into equivalence classes, with results that are close enough to each other in the same equivalence class. How close is close enough depends on the application.

The preliminary acceptance test classifies real-valued results as correct when they are within 0.1 of a reference value. Thus the equivalence class of correct results can include values that are up to 0.2 from each other. It was observed in the Second Generation Experiment that when versions agreed with each other to one or two decimal places, they often agreed much further. For consistency with the preliminary acceptance test, real-valued results are classified as equivalent when they agree to within 0.1.

For convenience, real-valued results are classified as correct when they are members of the equivalence class with the most members. The correct results are determined automatically by running all of the versions on the input and determining the equivalence class with the most members for each output variable. If the size of the equivalence class with the most members is less than half of the versions, then the correct result is determined by hand. The automatically generated correct results were manually checked to ensure that the values for all variables indicated a consistent situation. There was also a "gold" version available for checking the correctness of results from the Second Generation Experiment, but this analysis did not use it.

CHAPTER 2

Analysis

2.1 Unreachable Code

From 94.7% to 99.6% of the branches in each version were covered. 0.0% to 3.58% were illegal branches (See Figure 2), and the remaining 0.0% to 1.96% were unreachable branches (See Table 2). This section focuses on the unreachable branches.

Version	Unreachable Branches	Illegal Branches
UIUC1	0.39%	1.19%
UIUC3	0%	0.38%
UIUC4	1.96%	3.26%
UIUC5	1.42%	0%
NCSU1	1.93%	1.50%
NCSU2	0.65%	0.65%
NCSU3	1.41%	0.40%
NCSU4	0.34%	0%
NCSU5	1.47%	1.47%
UVA1	1.69%	0.18%
UVA2	0.23%	0.23%
UVA3	0.36%	0.73%
UVA4	0.63%	0.94%
UVA5	0.18%	1.50%
UCLA1	1.23%	2.46%
UCLA2	0.30%	2.43%
UCLA3	1.06%	2.40%
UCLA4	1.19%	3.58%
UCLA5	1.23%	0%

Table 2. Percent Unreachable and Illegal Branches

There were three common causes of unreachable branches: general-purpose modules, defensive programming, and redundantly nested **if** constructions. A general-purpose procedure is a procedure that solves a problem in greater generality. A general-purpose procedure often traps a variety of error conditions. However, if none of the calls to the procedure has one of the error conditions, it is not possible to cover the branch(es) that trap that error condition, and the branch is unreachable. For example, some of the versions have a general-purpose procedure to multiply two matrices. In these procedures, there is a check to see if the dimensions of the matrices are properly matched for multiplication. These procedures are called only with matrices with appropriate dimensions. Thus any branches pertaining to this case are unreachable.

Defensive programming is a programming style in which a programmer includes checks for error conditions in the code, instead of assuming that the error conditions will not occur. Many programmers included redundant checks for special conditions. For example, many of the versions check to make sure that the system is operational, i.e. at least four sensors passed a consistency check, before calling their procedure to estimate the acceleration. Yet these estimation procedures also check whether the system is operational. The checks inside the procedure can never find that the system is not operational.

The third common cause of unreachable code is nested **if** structures of the following form:

```
(1) if A or B
(2)     then if A
(3)         then $S_1$
(4)         else if B
(5)             then $S_2$
(6)             else UNREACHABLE
(7)     else $S_4$
```

The **if** statement on lines 4 to 6 is superfluous; the **else** on line 4 is covered exactly when B is true and A is false. None of the programmers actually wrote the **else**'s on lines 6 and 7, but the test for B on line 4 implies that it is possible for B to be false. It appears that the programmers did not look at their code hard enough to realize what they were doing.

None of these common causes of unreachable code seems to affect correctness. In fact, the first two tend to improve correctness. It is often easier to find a published general-purpose algorithm, or one from a library or another program, that is very trustworthy than to create a reliable algorithm from scratch, even if one can take advantage of the limitations of the particular application. Of course, it is possible to use these procedures incorrectly, but they are less likely to contain faults than starting from scratch.

It is good practice to write procedures that check that the conditions they require to perform correctly are true. If everything is put together correctly, the redundant checks are not necessary. However, when the system is first put together and whenever changes are made to the system, the extra checks are important safeguards. These checks may also alert the tester to other problems, e.g. an incorrect call of a procedure.

2.2 Illegal Branches

The specification used in the Second Generation Experiment stated that many conditions would not occur. Some of these statements were very reasonable and some were questionable. Nevertheless, inputs causing these conditions are invalid, and the branches that test for any of the conditions are illegal branches. The illegal branches were not tested.

The sensors are attached to the faces of the RSDIMU. One of the more reasonable restrictions put on the input was that the angles that measured how much the orientation of the sensors differed from their ideal orientation would be would be less than five degrees. It seems reasonable to assume that if these angles are too large, the RSDIMU will not be used. None of the versions check for an error condition caused by these angles being too large, evidently thinking that it is a reasonable restriction on the input.

The specification also includes restrictions on the input that are more questionable. For example, the specification stated that if a sensor failed in flight, it would fail unambiguously, i.e., that its reading would be very inconsistent with the readings from the working sensors. The algorithm that detects failed sensors uses a redundant set of consistency checks. A sensor reading may be inconsistent enough to fail some of these, but not all. The specification does not say how to classify a sensor in this ambiguous case, but instead says that the case will not happen. This is especially unreasonable, because it is physically possible for a sensor to "drift" from correct readings. When a sensor starts "drifting," it may pass only some of its consistency checks. Many programmers evidently did not believe the specification, and put in branches that deal with ambiguous results from the consistency checks.

2.3 Coverage Sets

The size of a coverage set is the number of test cases in the coverage set. The versions vary widely in the size of their coverage sets (See Figure 8). The execution of the first test case covers branches that get covered by the execution of almost any test case. Thus, the number of uncovered branches after the execution of the first test case is a better indicator of the amount of testing to be done. The preliminary acceptance test is treated as the first test case. The size of the coverage set tends to increase with the number of branches still uncovered after the preliminary acceptance test. However, there is a lot of variation in this relationship. This variation may be due to some versions having one or more fairly large sets of branches that can be covered by the execution of a single test case. This variation may also be due to poor selection of test cases for some versions: some attempts to develop a test case whose execution covers a particular branch fail. The attempt may fail for various reasons. The code around a branch may be hard to understand; the comments around a branch may be misleading; slightly wrong meanings may be suggested by variable names; the conditions necessary to enter the procedure in which a branch is found may not be considered at first.

There is a common pattern in developing coverage sets (See Table 3). It takes several iterations of examining branches, developing new test cases, and running a version on the new test cases to cover that version. Usually the first one or two iterations have a lot of test cases. Then the number of test cases in an iteration tapers off, because there are fewer branches left to cover. The branches covered toward the end of testing are more difficult either because the conditions necessary to cover them are hard to understand, or because the test cases whose execution would cover them are hard to create. Some of the versions, e.g. NCSU3, have a small first iteration before following the usual pattern, because those versions have procedures with many branches in them that were not called during the preliminary acceptance test. During the first iteration, test cases are developed whose executions cover a branch that calls a procedure with many branches. On the next iteration, after some of the branches have been covered, many test cases are developed whose execution covers the remaining branches.

2.4 Independence of Testing

One person could oversee the ordinary branch coverage testing of all versions, or each version could be tested by a different person, for example, the programmer. Both methods have advantages and disadvantages.

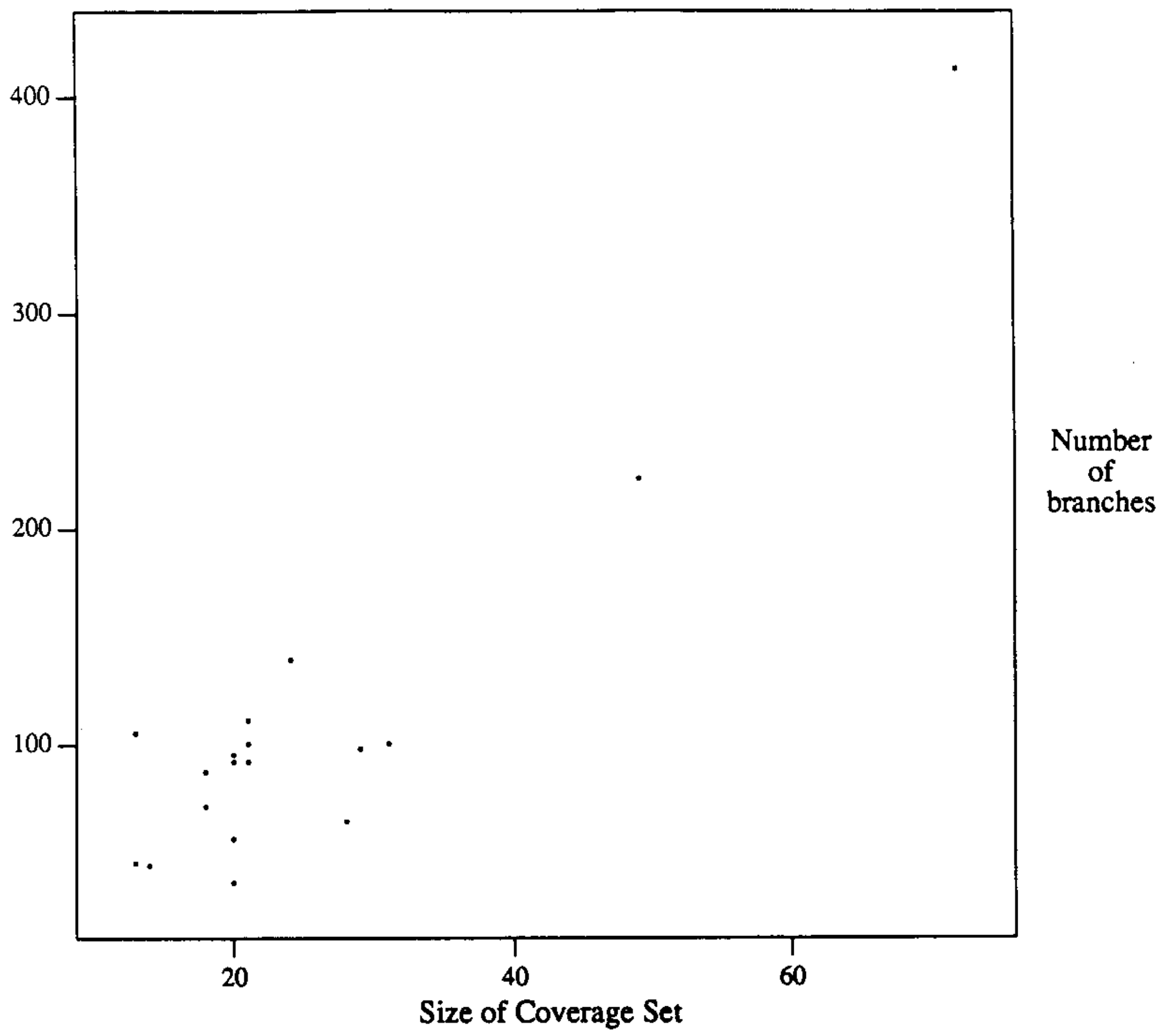

Fig. 8. Size of Coverage Set versus the Number of Uncovered Branches after Preliminary Acceptance Test

If one person does the testing, he profits from an understanding of the system as a whole. There is a consistent view of correct results. Inconsistent interpretations of the specification may be uncovered and therfore corrected earlier than with different testers. The tester gains insight into the problem and the relative strengths of

	Num Cases in Iteration						Size	Num
Version	1	2	3	4	5	6	Cover. Set	Iterations
UIUC1	10	8					18	2
UIUC3	20	5	2	1			28	4
UIUC4	3	4	3	6	3	1	20	6
UIUC5	9	3	1				13	3
NCSU1	9	11	1				21	3
NCSU2	2	9	7	3			21	4
NCSU3	5	23	10	5	4	2	49	6
NCSU4	2	2	5	4	1		14	5
NCSU5	10	12	7				29	3
UVA1	16	8					24	2
UVA2	13	4	1				18	3
UVA3	33	31	7	1			72	4
UVA4	5	2	2	11			20	4
UVA5	11	18	2				31	2
UCLA1	4	4	6	3	3	1	21	6
UCLA2	8	1	6	2	3		20	5
UCLA3	3	2	4	4			13	4
UCLA4	5	1	8	2	5		21	5
UCLA5	3	8	5	3	1		20	5

Table 3. Number of Test Cases in each Testing Iteration.

each version because he has to understand a large part of each version. This insight allows the tester to develop test cases that exercise the versions more thoroughly than with this insight divided among several people.

If each version is tested using ordinary branch coverage testing by a different person, then there is less chance of a common fault being overlooked during the testing phase. If the programmers test their own versions, then they will have a much easier time understanding the code and the conditions for covering branches than anyone else. If the testers are not the programmers, then the testers are less likely to be influenced by what they expect the code to do because they did not write it or see similar code in other versions. However, it is possible for the testers to disagree on what the correct results are. If this happens, it will be discovered at the next stage, when all versions are tested on all test cases. There are likely to be branches in different versions that need the same or nearly the same conditions in order to be covered. With several different testers developing test cases for these branches, they will be tested more variously than if one tester were developing all of the test cases. However, it is not clear that these frequent branches are where the testing effort should be concentrated.

2.5 Evaluation of Group Branch Coverage Testing

2.5.1 Comparison with Ordinary Branch Coverage Testing

Group branch coverage testing is expected to reveal more faults than ordinary branch coverage testing. Group branch coverage testings designed to find all of the faults found by ordinary branch coverage testing by using the union of the versions' coverage sets. For these versions and the coverage sets developed for them, almost 40% additional faults were found using the test cases developed for the other versions.

The shaded part of the bars in Figure 10 shows the number of faults found for each version by its coverage set alone. The unshaded part of the bars shows the number of additional faults found by group branch coverage testing. The column of numbers to the right of the bars shows the percentage of the faults found by group branch coverage testing, that were not found by ordinary branch coverage testing. There is wide variance in how much the group part of the testing helped individual versions, from 0% for UIUC4 and UVA1 to 100% for UCLA1. This variation may be due to variation in the degree to which the versions were tested by their programmers.

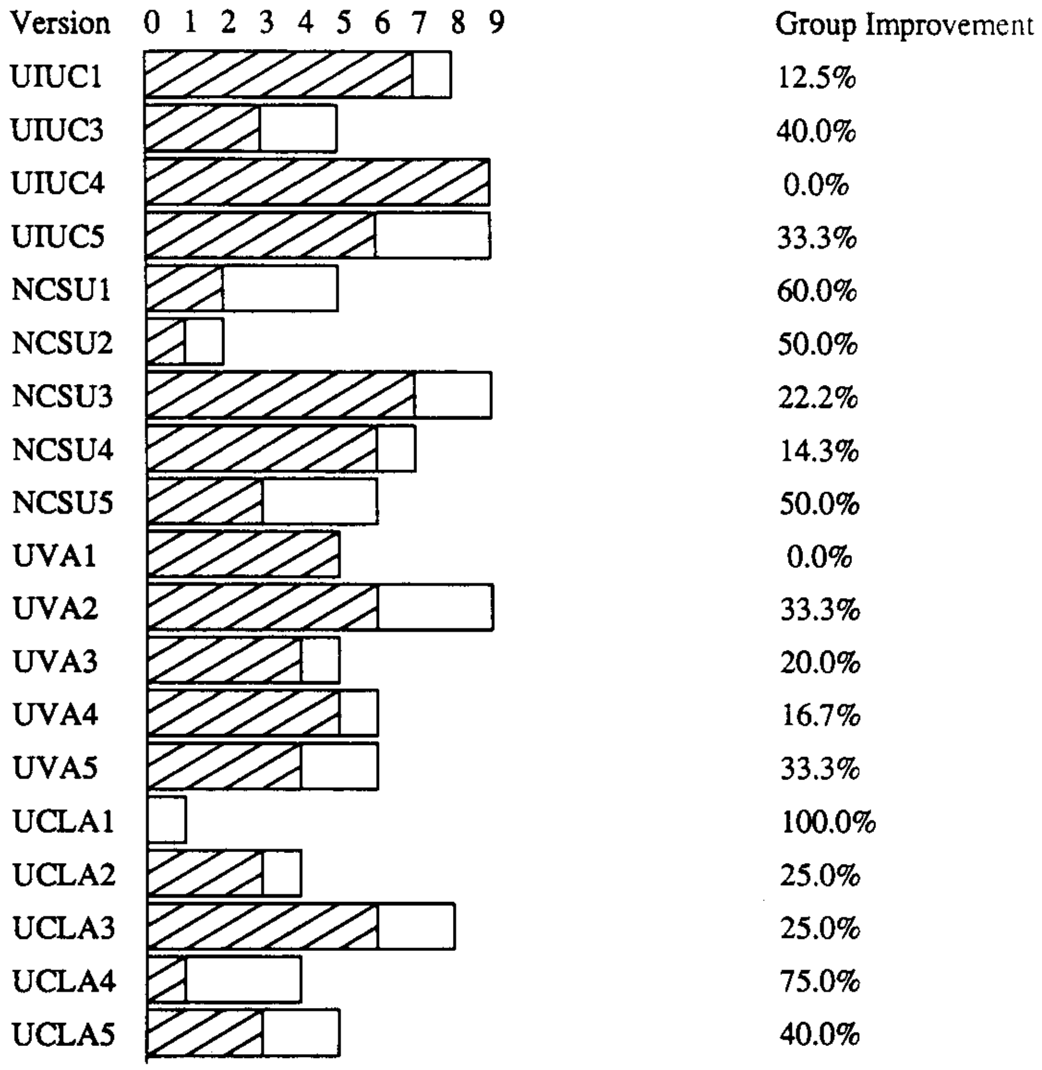

Fig. 9. Number of Faults Found.

If the programmers tested their versions at least as thoroughly as branch coverage testing, then another pass of branch coverage testing is unlikely to reveal many more errors.

2.5.2 Comparison with a New Acceptance Test

A number of faults were found in the UCLA versions during other testing. The specification used in the Second Generation Experiment has been corrected. The five UCLA versions have been modified to adhere to the new specification; [Kel86] the other versions are being modified at the universities at which they were produced. As the UCLA versions were modified and they were tested by a new acceptance test, faults were uncovered in each of them (See Table 4). Many of these faults found had not been revealed by the group branch coverage testing.

Version	Coverage Set	Union	New Acceptance Test	Total	Overlap
UCLA1	0	1	5	6	0
UCLA2	3	4	2	6	1
UCLA3	6	8	9	17	4
UCLA4	1	4	8	12	1
UCLA5	3	5	3	8	1
Total	13	9	27	49	7

Table 4. Number of Faults Found in UCLA Versions

Two-thirds of the faults found only by the new acceptance test concern either the proper placement of decision points for recovery, or computational errors whose effect on real-valued variables is less than 0.2. Under recovery, the consensus decision is passed back to the versions from the decision algorithm. The faults related to the decision points that were found by the new acceptance test were of two sorts: in some versions the decision algorithm was called after the variable had already been used in some computation; in others, the changed vaules of returned output variables were ignored.

Most of the faults found by group branch coverage testing of the UCLA versions concern either the display module or the computation of boolean-valued variables. The new acceptance test did not reveal faults in the computation of boolean-valued variables, because the definitions of these variables were changed during the modification of the specification.

The new acceptance test revealed some subtle computational faults that group branch coverage testing did not, because the new acceptance test used much smaller equivalence classes to determine correct results. During the group branch coverage testing, wrong results were considered correct if they differed by the correct results by less than 0.2. With a better method of determining whether a result is acceptable, group branch coverage testing would have revealed some of these less obvious computational faults.

It is not surprising that group branch coverage testing did not reveal faults in the placement of the decision points. The decision points must be executed (See Figure 6); it is unlikely that they would be placed in a conditional statement. Testing versions to see whether they call the decision algorithm properly for recovery involves modifying the decision algorithm so that it returns values different than those that were passed to it, but still consistent with the rest of the variables so that the computation can continue. Branch coverage testing does not enforce this kind of testing.

2.6 Similar Errors

The multi-version approach to fault-tolerance depends on the low probability of similar errors in the system. In a three version system, the decision algorithm will produce an erroneous result in the presence of any similar failures of the versions. An estimate of the proportion of faults that lead to similar errors in versions is useful for evaluating MVS systems.

The faults found by group branch coverage testing can be used to empirically estimate the proportion of faults that produce similar errors. It is recognized that this estimate is valid only for these 19 versions of this RSDIMU software. However, it adds to the meager empirical results from testing MVS systems and their components. A larger collection of empirical results will allow a more realistic assessment of how much reliability improvement can be expected from MVS systems.

Fifty-nine faults were found. A fault can be manifested in more than one version. The number of versions in which a fault occurs is called the fault's *occurrence number*. A fault with a higher occurrence number is more likely to be in a randomly selected version and to be in enough versions that the decision algorithm cannot make the correct decision. The faults' occurrence numbers varied a lot. Most faults have an occurrence number of one, but one fault occurred in eight versions. Figure 11 shows the number of faults with each occurrence number.

The versions themselves were used to classify correct results automatically. If a fault occurred in ten or more versions, the errors from that fault would have been classified as correct by the automatic classification. Faults were found manually. The results from all versions for all output variables of an input case were examined. For a particular version, all output variables that were classified as incorrect were noted. For each incorrect output variable, the code of that version dealing with that output variable was examined until the fault was found. If a correct result was incorrectly classified as incorrect by the automatic classification procedure, then the mis-classification would be detected by the supposed fault that produced the correct result. In this way, the only mis-classification of incorrect results as correct that would not be

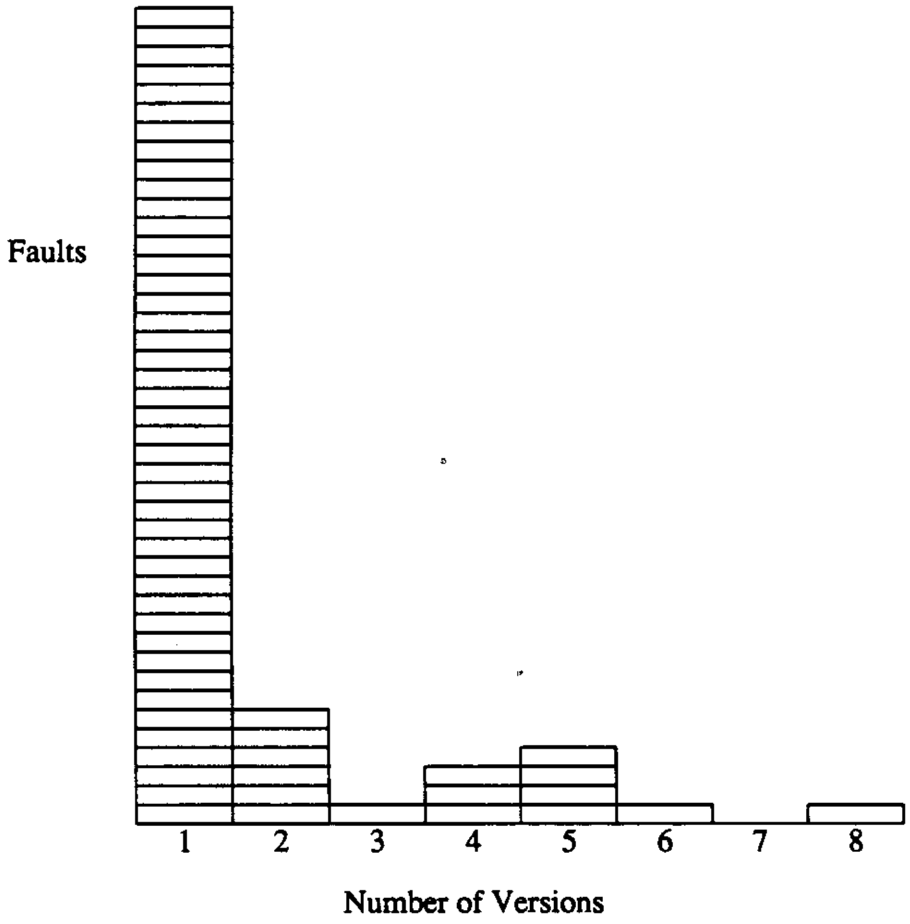

Fig. 10. Occurrence of Faults

detected are those for which no correct result for that output variable was produced for any of the 19 versions. This mis-classification occurs when a similar error in several versions is incorrectly classified as correct, and none of the rest of the versions are correct either; since there is a fault in all of the versions whose outputs are labeled incorrect, there is no reason to suspect that the result labeled correct is wrong. Thus, it is reasonable to assume that there were not faults that occurred in more than ten versions that were not detected because the versions were used to define correctness.

It has been shown that the specification may be responsible for some of the similar faults [Kel86, Avi84]. The distribution of the occurrence numbers of similar faults that were and were not related to the specification are shown in Figures 11 and 12, respectively. The specification-related faults tend to have high occurrence numbers. Since all programmers used the same specification, it makes sense that the faults that derived from the specification occur in many versions. For example, the fault with the highest occurrence number, 8, is the computation of whether a sensor is noisy or not, even though the sensor is already known to be failed. The specification, however, accounts for only a quarter of the similar faults (See Appendix B for a list of all the similar faults found in this research).

The other faults appear to be attributable to programmer oversight or inability to consider the entire specification at once. For example, the fault with the second highest occurrence number, 6, is failure to protect against division by 0. The versions with this fault abort whenever there are no operational sensors. Another example of a common fault, occurrence number of 5, is the computation of output variable "LINOFFSET" for a sensor that is known to be failed at the beginning of the computation. The specification clearly states that "LINOFFSET" should be set to a default value when the corresponding sensor is known to be failed. Evidently many programmers forgot this detail of the specification. Since the preliminary acceptance test did not check these basic details of the specification, the detection of these faults was left to later testing.

2.7 A Detailed Example

The behavior of the versions' display module on a test case is examined in detail. The test case was selected because of the many faults it reveals in the display module. It is not typical of the density of faults revealed by one test case. The ver-

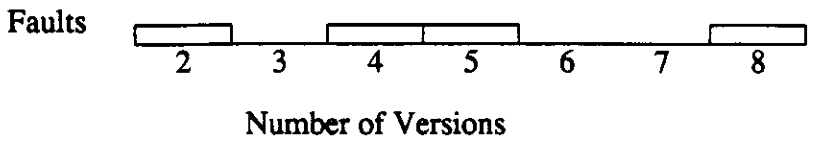

Fig. 11. Occurrence of Specification Related Similar Faults.

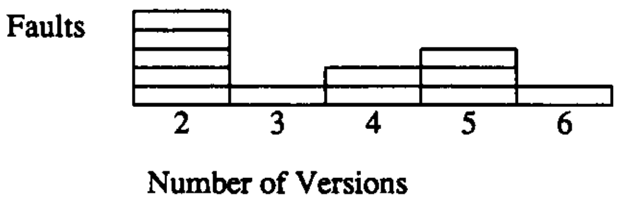

Fig. 12. Occurrence of Non-Specification Related Similar Faults.

sions are supposed to compute an acceleration of <+9.999992, +10.00499, +10.00153> and then display the X-component of the acceleration to five significant digits: "+10.000".

There are three words (integers) used to drive this row of the display panel (See Figure 6). The first word hold the bits that display the first two digits. The second word hold the bits that display the next two digits. The last word hold the bits that display the last digit, the decimal point, and the sign. Each bit in the word controls one segment.

The correct result for each word was separately determined by consensus. Nine different display patterns were generated by the 19 versions – and four versions did not generate display patterns because they aborted (See Table 5). In fact, no version agreed with the automatically classified "correct" result on all three words! The automatically classified "correct" display was "+1.0000". Seven versions, UIUC1, NCSU1, UCLA5, UVA1, UIUC4, UIUC5, and UVA4, formed the consensus group with the most members saying the first two digits were "10". There was almost complete agreement that the next two digits, the second word, were "00". For the last word, the consensus group with the most members had seven members, NCSU4, UVA2, UVA3, UVA5, NCSU2, UCLA4, and UCLA1, saying the last digit was "0", the decimal point was after the first digit, and a plus sign should be displayed.

Display	Versions Displaying It
+10.000	UIUC1, NCSU1, UCLA5
+A.0000	NCSU4, UVA2, UVA3, UVA5
+0.0000	NCSU2, UCLA4
+00.000	NCSU5
10.000	UVA1
+.1.00.0.0.	UIUC4
+9.0000	UCLA1
+10.005	UIUC5
+10.274	UVA4
abort1	NCSU3, UCLA2
abort2	UIUC3, UCLA3
+1.0000	correct result, determined automatically

Table 5. Displays of Acceleration +9.999992.

The problem in displaying a real number is breaking the number down into digits so that the appropriate bit pattern to display the digits can be determined. Because the number is rounded before displaying, some of the digits may be different than the digits in the original number. The general approach to this display module involves first finding the magnitude of the number, i.e. where the most significant digit is relative to the decimal point. Once the magnitude of the number is determined, the number can be rounded to five significant digits. After rounding and determining the magnitude, the digits can be isolated easily, using integer division and mod (the "remainder" function). Rounding +9.999992 to five digits changes the location of the most significant digit from the one's place to the ten's place.

Two of the versions, UIUC5 and UVA4, compute different accelerations whose x-components are both larger than 10. They both correctly display the acceleration they compute, but do not encounter the problem the other versions do in rounding the acceleration. Only the remaining 17 versions are considered in the rest of this discussion.

Only six of the versions account for the change in the position of the most significant digit after +9.999992 is rounded. Of these, two, UVA1 and UIUC4, had other problems with the display: UVA1 does not display the plus sign; UIUC4 displays the decimal points it means not to display and does not display the decimal point it does mean to display. Another, NCSU5, displayed the digits in the one's place to the 0.0001's place as the five most significant digits, instead of the digits from the 10's place to the 0.001's place. Thus, only three versions correctly computed the display.

"+A.0000" was the most common display. It is produced by treating the integer part of the number, 10, as the first digit and displaying it. It is sometimes required of the versions to display hexadecimal numbers (numbers in base 16). In the hexadecimal system, the digit ten is represented by "A". Thus the "digit" 10 was display as "A".

abort1 was caused when two versions treated the integer part of the number, 10, as a digit. This first "digit" was assigned to a variable that can hold only decimal digits, i.e., 0 to 9. Thus these two versions aborted.

"+0.0000" was produced by versions NCSU2 and UCLA4. It is produced by noticing that the most significant digit, before rounding, is in the one's place. These versions continued to treat the one's place as the most significant place, even after rounding. Thus they displayed "+0.0000", ignoring the 1 in the ten's place.

One additional version, UCLA1, found the first digit as it was finding the magnitude. It did not recalculate the first digit after rounding. It displayed "+9.0000".

UIUC3 and UCLA3 both aborted on this input, abort2. Both of these versions found the five most significant digits to be displayed, by multiplying the number by 10 until there were five digits to the left of the decimal point. The variable that held these five digits could not hold more than five digits. When rounding took place, six digits, i.e. 100000, were assigned to the variable. Thus the versions aborted.

CHAPTER 3

Conclusions

Group branch coverage testing found 40% more faults on the average than ordinary branch coverage testing. These additional faults tended to be more subtle than those revealed by ordinary branch coverage testing. More research is needed to determine how many more faults are detected for systems involving fewer versions.

Three-version software systems tolerate errors in one version, but they cannot tolerate similar errors. Therefore it is much more important to detect similar faults than distinct faults. Some of the faults detected by ordinary branch coverage testing were found to exist in other versions as well. Group branch coverage testing found these similar faults, as well as those for which no occurrence was detected by ordinary branch coverage testing.

This research verifies that similar faults may be related to the specification. However, more than half the similar faults were not related to the specification. If the preliminary acceptance test had used better criteria for correctness, i.e. had used smaller equivalence classes for correct values and had checked that all output variables be correct, then many of the similar faults would have been revealed during it. Better correctness criteria would have revealed subtle computational faults. It is likely that inclusion of these subtle computational faults would affect the proportion of similar fault in the faults found, but it is difficult to predict how.

Up to 5% of the code in some versions was uncoverable. Testers should not be surprised when they find uncoverable code. The most common causes of branches being unreachable were general-purpose procedures and defensive programming. A thorough understanding of the code is necessary in order to be sure that a branch really is unreachable. The most common cause of illegal branches was programmers distrusting the specification enough to test for conditions that the specification stated would not occur. Illegal branches point up concerns that the specification addressed inadequately, or not at all. Looking at the conditions in the illegal branches helps to validate the specification.

There are advantages to both having one person test all versions and having all versions tested independently before they are tested on the union of their coverage sets. Further research is needed to show the affects of multiple testers. One tester gains deeper insight from his familiarity with all the versions. A consistent definition of correct is provided by one tester. The main advantages of multiple testers are that more test cases are developed and there is less chance of introducing a common fault through testing.

References

[Adr82] Adrion, W. Richards, Martha A. Branstad, and John C. Cherniavsky, ''Validation, Verification, and Testing of Computer Software,'' *ACM Computing Surveys* **14**(2), pp.159 - 192 (June 1982).

[Avi84] Avižienis, A. and J.P.J. Kelly, ''Fault-Tolerance by Design Diversity: Concepts and Experiments,'' *Computer* **17**(8), pp.67-80 (August 1984).

[Bis85] Bishop, P., D. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti, and S. Yoshimura, ''Project on Diverse Software - An Experiment in Software Reliability,'' *Proceedings IFAC Workshop SAFECOMP'85* (October 1985).

[CRA85] CRA, ''Redundancy Management Software Requirements Specification for a Redundant Strapped Down Inertia Measurement Unit,'' Version 2.0, Charles River Analytics and Research Triangle Institute (May 30, 1985).

[Eck85] Eckhardt, D.E. and L.D. Lee, ''A Theoretical Basis for the Analysis of Redundant Software Subject to Coincident Errors,'' 86369, NASA, Hampton, Virginia (January 1985).

[Hil85] Hills, A.D., ''Digital Fly-By-Wire Experience,'' *Proceedings AGARD Lecture Series*(143) (October 1985).

[Kel86] Kelly, J.P.J., A. Avižienis, B.T. Ulery, B.J. Swain, R.T. Lyu, A.T. Tai, and K.S. Tso, ''Multi-Version Software Development,'' pp. 43-49 in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France (October 1986).

[Mar82] Martin, D.J., ''Dissimilar Software in High Integrity Applications in Flight Controls,'' pp. 36.1-36.13 in *Proceedings AGARD-CPP-330* (September 1982).

[Pra83] Prather, R. E., ''Theory of Program Testing – An Overview,'' *The Bell System Technical Journal* **62**(10, Part 2), pp.3073-3105 (December 1983).

[Ram81] Ramamoorthy, C.V. and *et al.*, ''Application of a Methodology for the Development and Validation of Reliable Process Control Software,'' *IEEE Transactions on Software Engineering* SE-**7**(6), pp.537-555 (November 1981).

[Tso86] Tso, K.S., A. Avižienis, and J.P.J. Kelly, "Error Recovery In Multi-Version Software," pp. 35-41 in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France (October 1986).

[Vog85] Voges, U., "Application of a Fault-Tolerant Microprocessor-Based Core-Surveillance System in a German Fast Breeder Reactor," *EPRI-Conference* (April 9-12 1985).

[Vou86] Vouk, Mladen A., Michael L. Helsabeck, Kuo-Chung Tai, and David F. McAllister, "On Testing Functionally Equivalent Components of Fault-Tolerant Software," pp. 414-419 in *Proceedings COMPSAC the IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference*, Chicago, IL (October 8-10, 1986).

Appendix A

This is the trace file produced when the sample tax program is run on the sample input. It is intended to be analyzed by a coverage analyzer, not for a human being to read. The "N"'s indicate that a procedure has been entered. The first number on an N line is the number of letters in the name of the procedure. The second number is the number of branches in the procedure. The lines that are indented one space, give the names of procedures. The "U"'s probably indicate that a branch has been covered.

```
N 3  9
 tax
U 1  1
N 8  2
 GetInpu
 t
U 1  1
U 1  2
R 8
 GetInpu
 t
U 1  3
U 1  4
N 10  3
 Compute
 Tax
U 1  1
```

```
U 1  2
R 10
 Compute
 Tax
U 1  9
R 3
 tax
```

Appendix B

Similar Errors	Occurence Number
Does not set all sensors to failed when RSDIMU not operational	4
Does not set all sensors to failed when RSDIMU becomes non-operational in-flight	5
Shows 1's place for nums with |num| < 1	2
The sensors' readings are recorded in 16-bit words and the highest 4 bits should be ignored. High bits not ignored for display purposes	5
High bits not ignored in computations	5
Divides by 0 when all sensors are failed	6
Integer variable is supposed to hold 5 most sigificant digits. Number is multiplied by 10 until there are 5 digits to left of decimal point. The type of the variable prevents 5 digit numbers starting with 7, 8, or 9 from being assigned to it.	2
Value of "LINOUT[i]" is set to special value whenever sensor i is noisy. Should only be done for sensors previously failed	4
Tests previously failed sensors for noise	8
Threshold for determining whether a sensor without an operational partner has failed is too low	3
Previously failed sensors always labeled noisy	2
Displays "+0.0000" for acceleration component = +9.999992	2
Multiplies acceleration component by 100,000, causing an "Integer overflow" abort when the acceleration component is too big	2
Aborts when +9.999992 is rounded in display; attempts to treat ten as a digit	2
Displays "+A.0000" when acceleration component is +9.999992	4
Computes value for variable "LINOFFFSET[i]", even when sensor i is known to be failed	5

Similar Errors Found During Group Branch Coverage Testing