Software Fault Localization with Theory of Evidence

by

Adam Jordan, B.S.

A Thesis

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Dr. Rattikorn Hewett
Chairperson

Dr. Michael Shin

Dr. Yuanlin Zhang

Ralph Ferguson
Dean of the Graduate School

December, 2010

ACKNOWLEDGMENTS

I would like to thank Dr. Rattikorn Hewett for her guidance throughout my Master's research. Her broad field of knowledge was invaluable and her teachings will be beneficial to me long into the future. I would also like to thank Dr. Phongphun Kijsanayothin for his valuable comments and assistance in the development of this thesis.

I would also like to acknowledge the University of Nebraska at Lincoln for their distribution of the Siemens program suite for testing of fault localization techniques

TABLE OF CONTENTS

ABSTRACT

Software development is a worldwide business that affects almost all aspects of our lives. In the cycle of software development, software debugging is the most time consuming phase and in debugging, the process of locating software faults takes the majority of the time. The process of automating fault localization is a valuable asset to any development of large scale software. The larger an application scales, the more complex it becomes, and the more difficult it becomes to manage and locate faults within the software.

Automated software fault localization is used to try to locate a fault with little or no human intervention. In the past, this has been accomplished by analyzing test cases, execution sequences, logical predicates, memory states, and various other methods. This thesis presents a new technique of automated software fault localization that is based on theory of evidence for uncertainty reasoning to estimate likelihoods of faulty locations. The proposed technique is evaluated and compared to the three best-performing methods presently available using a set of benchmark programs in an empirical study. The study compares the methods' abilities to reduce the amount of code that needs to be viewed to locate a fault. The results show that the proposed technique performed no worse than these top techniques in 100% of all the program versions in the benchmark set with an average of over 85% of effectiveness measure.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The development of software is a large scale business that affects almost all aspects of our lives. Software exists in almost every electronic device we come in contact with from mobile phones, to video games, e-readers, and to everyday applications we use on the personal computer. According to DataMonitor, a market researcher, software development was a $303.8 billion business in the United States in 2008, and it is projected to be $457 billion business in 2013 (Popp and Meyer, 2010).

The software development process consists of a number of phases. These phases include design, implementation, testing, deployment, and maintenance. Software debugging often plays a major role in several of these phases. When software does not perform according to its specifications, software debugging is required to identify a problem and fix it. The process of debugging consists of a number of steps, which would include first verifying that a problem exists, then locating the problem, and finally, fixing the problem.

Following the conventional terminology (Abreu et al., 2007), a software *failure* is defined to be an event that occurs when software output for a given input deviates from the specified output for that input. Software execution may have *errors* that may or may not cause a failure. Thus, it is possible that these error states of software execution may not be found. A software *fault* (or a *bug*) is the cause of an error. A fault in a program may be an incorrect line of code, a missing statement, or anything else that causes the logic of the program to execute incorrectly and proceed to become an error.

Finding where a fault occurs within a program is referred to as *fault localization*. Current techniques for fault localization often involve a manual process, which can take a variety of forms. A developer may get a memory dump to try to understand the state of the program when a failure occurs. The developer may add a

number of logging statements using any number of methods, such as file output or console streaming, to understand the control flow and state of the program. Fault localization may also consist of watching a program's execution in a symbolic debugger, using break points, and controlling the execution. In this case, the developer would look at the program state, step in and out of code blocks, and try to locate the fault manifests. Debugging software is known to be the most time consuming and costly step of software development (Vessey, 1986). As software being developed becomes larger and more complex, techniques for automating fault localization to assist debugging become increasingly more important.

Recent research has studied various approaches to software fault localization. The most common approach, *spectrum-based fault localization* (Abreu et al., 2007), locates faults within a program by analyzing *program spectra*, which can be described as follows. In software testing, when a program's execution output strays from the expected output, this is referred to as a *failed execution*, or *failed test*. When the execution output matches the expected output, this is a *successful execution*, or *passed test*. During an execution, a set of program unit, such as lines and blocks of code, is executed. This information coupled with the result of the execution is defined as a set of *program spectra* (Reps et al., 1997). Spectrum-based fault localizers have shown to be more effective than other techniques (Abreu et al., 2006), such as diagnostic reasoning (Sedlmeyer, 1983), program slicing (Weiser, 1982), Nearest Neighbor (Renieres and Reiss, 2003), statistical approaches (Liblit et al., 2005; Liu et al., 2005), and memory analysis (Zeller and Hildebrandt, 2002; Cleve and Zeller, 2005)

There are a variety of spectrum-based fault localizers (Jones et al., 2002; Chen et al., 2002; Meyer et al., 2004), each of which differs by its *similarity-coefficient* for identifying a software unit in the spectra that is the most similar to the test results indicating whether each of the execution runs succeeds or fails (Abreu et al., 2006). The top performers of spectrum-based fault localizers include *Ochiai* (Meyer et al., 2004), *Jaccard* (Chen et al., 2002), and *Tarantula* (Jones et al., 2002). The first two

methods are named after two similarity-coefficients and the last is based on a heuristic function.

This thesis proposes a new spectrum-based technique for software fault localization that is based on the theory of evidence called Dempster-Shafer Theory (Shafer, 1976). Specifically, the thesis develops mass functions for estimating likelihoods of faulty locations based on evidence from execution runs obtained from the program spectra. Unlike other existing spectrum-based techniques, the proposed technique is well grounded by a widely used mathematical theory. We evaluate the proposed technique and compare it with the three best state-of-the-art methods: Ochiai, Jaccard, and Tarantula. This study is performed based on a set of standard benchmark programs. The main contributions of this thesis are as follows.

(1) A new spectrum-based method of automated fault localization that is based on a theoretically grounded work for reasoning under uncertainty.
(2) An empirical study and a comparative analysis between the three top methods of fault localization and the proposed method.

The rest of this thesis is organized as follows. Chapter II discusses the work related to fault localization and the proposed method. Chapter III describes preliminary concepts and terminology including basic mechanisms for software fault localization, the Dempster-Shafer Theory and its fundamental elements, and the three existing spectrum-based fault localizers used in our comparison study. Chapter IV presents our proposed method for software fault localization and illustrates the method in a small example program. It also describes some characteristic comparison between our proposed method and the others. Chapter V evaluates the proposed technique and discusses the empirical study using a set of standard benchmarks to compare the proposed method against the other three prominent software fault localization

methods. Chapter VI gives concluding remarks and possible extension for future work.

CHAPTER II

RELATED WORK

The area of automated software fault localization is not a new field and has a long history of methods for locating faults.

Diagnostic reasoning, an expert system built in order to use rules to locate a fault (Sedlmeyer et al., 1983) was an early method. This required contextual information to describe what typical program behavior would be in order to understand when it deviates. This required knowledge of the underlying system.

Another early method was the slicing technique, which tried to break down a program into smaller segments in order to lessen the locations where a fault may exist (Weiser, 1982). This was done by certifying that a variable should have a specific value at a given point in time, if it did not, you could assume that the fault occurred prior to that point. The problem with this is that the variable, statement pairs are set up independent of individual tests, so you cannot link a specific failed test with a variable value.

More recently, spectrum-based fault localizers have been developed, such as Tarantula, Jaccard, and Ochiai which are used in the empirical study and described in a later section. This group also includes the Nearest Neighbor technique (Renieres and Reiss, 2003), which tried to contrast the statements executed in a failed test to that of a successful test by contrasting the distance between the two. Therefore if a statement appeared in a failed test, but not a successful test, then it was likely to be a fault. If the faulty line did not exist in this set, then a code dependence graph was constructed and followed until the fault was found. This was also more generalized into the set union and set intersection techniques. The primary problem with this approach was that it did not have a tolerance for faulty statements present in a successful test because the dependence graph was too inefficient (Jones and Harrold, 2005).

The statistical approach is another method of fault localization (Liblit et al., 2005; Liu et al., 2005). These methods analyze the code by use of predicates, or logical values, during the running of a program. The issues with these methods is that they rely on good predicates for a given source code base and located faults are limited to those located within the predicates.

There are some methods that monitor variables and program state to try to locate faults. The Delta Debugging (Zeller and Hildebrandt, 2002) technique tried to analyze the state of variables in both successful and failed tests to distinguish which part of a program a fault may exist in. This was done by using memory graphs. The Cause Transitions (Cleve and Zeller, 2005) method extended Delta Debugging by doing a binary search of program state to narrow the possible locations of the fault. This search consisted of trying to automate the process of finding where a specified variable changed state. These methods proved effective, but are very resource intensive and still did not perform as well as Tarantula (Jones and Harrold, 2005).

There has also been more recent work using machine learning techniques to automate the locating of faults. Wong et al. (Wong et. al, 2009) proposed a fault localization technique based on back-propagation neural networks, which used the program spectra to train the neural network and virtual test cases for each segment of code as input into the network. This suffered from traditional problems, such as paralysis and local minima. This work was extended to use radial basis function networks (Wong et al., 2008), which was less susceptible to these problems.

CHAPTER III

PRELIMINARIES

This section outlines the foundation for the proposed method of fault localization in Dempster-Shafer theory, introduces the selected methods that will be used in the empirical study, and explains the underlying concepts in program spectra and similarity coefficient rankings.

Theory of Evidence: Dempster-Shafer Theory

To provide a theoretical background of the proposed research, we describe the mathematical theory of evidence, particularly, the *Dempster-Shafer Theory* (Shafer, 1976). Dempster-Shafer theory, also known as the theory of belief functions or theory of evidence, was developed by A. P. Dempster and Glenn Shafer in 1976. While a similar concept can be found back as early as the seventeenth century, it was not until the early 1980's when Bayesian probability theory was being applied to rule-based expert systems did it catch the attention of Artificial Intelligence (AI) researchers. By explicitly representing conditional probabilities in Bayes Rules, Bayesian probability theory provides core mechanisms to support inferences on uncertain events (Pearl, 2000).

Dempster-Shafer theory is a generalization of Bayesian probability in that it allows probability assignment to a *set* of atomic elements rather than an atomic element. Let U be a finite set of all hypotheses (atomic elements) in a problem domain. For example, the question being asked in the area of fault localization is whether a given line is faulty or not. Thus, in this problem domain, if a represents the hypothesis that the line is faulty and b represents the fact that the line is not faulty, the set of possible hypotheses, $U = \{a, b\}$. A *mass function m* provides a probability assignment to any $A \in U$. For example, for $U = \{a, b\}$ as defined previously in our problem domain, we have:

$$m(\emptyset) = 0$$

$$m(a) \in [0,1]$$

$$m(b) \in [0,1]$$

$$m(U) \in [0,1]$$

$$\sum_{A \in 2^U} m(A) = 1$$

Note that the first and the last formulae follow from the meaning of the mass function and the property of probabilities from the probability theory. Thus, they are true regardless what *U* is. When there is no information regarding *U*, $m(U) = 1$ and $m(A) = 0$ for all $A \epsilon U$. The former deals with a state of ignorance since the hypothesis set includes all possible hypotheses and therefore, its truth is believed to be certain.

In Dempster Shafer theory, every mass function has an associated *belief* function and *plausibility* function to respectively measure the degree of belief and plausibility of any subset of atomic elements (hypotheses) in *U*. Formally speaking, the degree of belief on *A*, $bel(A)$ and the plausibility of *A*, $pl(A)$ is defined below. For any $A \epsilon U$,

$$bel(A) = \sum_{X \subseteq A} m(X)$$

$$pl(A) = 1 - bel(\sim A) = \sum_{X \cap A \neq \emptyset} m(X)$$

For example, $bel(\{a\}) = m(\{a\})$. Thus, in general when *A* is a singleton set $bel(A) = m(A)$ and in such a case the computation of *bel* is greatly reduced. This is exactly the case in our problem domain and therefore the likelihood we estimate for

the hypothesis $\{a\}$, or $\{b\}$ from the mass function actually reflects the degree of belief in each case of this single hypothesis. Chapter IV will discuss the details of how we define the mass function for the fault localization problem.

One can see that each subset of all possible hypotheses may be supported or unsupported by one or more evidences. A mass function can be associated with evidence that contributes to a belief (or disbelief) of the given subset of hypotheses. Thus, there is a need to combine mass functions when there are multiple evidences. To do this, we apply the Dempster's combination rule as shown below. For $X, A, B \subseteq U$, a combination rule of mass functions m_1 and m_2, denoted by $m_{1,2}$ is defined as the following:

$$m_{1,2}(X) = m_1(\mathrm{X}) \oplus \mathrm{m}_2(X) = \frac{\sum_{A \cap B = 2^U} m_1(A) m_2(B)}{1 - K}$$

$$K = \sum_{A \cap B = \emptyset} m_1(A) m_2(B)$$

This rule emphasizes the agreement between multiple sources of evidences and ignores the disagreement from conflicting evidence through a normalization factor, 1 – *K*. Dempster's rule of combination can be applied pair-wise repeatedly across a set of evidences to produce a final belief for a given set of hypotheses. For more details of the Dempster-Shafer Theory, see (Shafer, 1976).

Coefficients, Ranking, and Program Spectra

To generate program spectra from a program's execution, we run a set of test cases and observe execution results. The test can performed at various levels of software units (e.g., lines of code, blocks of code, or some other subset of the code base). In our study, we use lines of code. In practice, it is also common to analyze the test cases in terms of blocks of code (Abreu et al., 2006; Nessa et al., 2008).

In most existing spectrum-based fault localization techniques, each software unit of a program is ranked based upon its corresponding similarity-coefficient value (Abreu et al., 2006) which measures how closely the execution runs of the test cases that involve the considering unit (as collected in the program spectra) resemble the errors observed from the output of each test execution. This is because it is assumed that the software unit that has a high similarity to these errors reflects a high probability that the software unit would be the cause of such errors. Thus, a unit with a larger similarity-coefficient value would rank higher in terms of its chance to be faulty. Unlike most existing spectrum-based techniques, in this thesis, our proposed technique does not compute the similarity-coefficient value of each line, but directly estimates the likelihood of it being faulty based on the evidence collected from the program spectra. Let S_i represent a similarity coefficient for a given code line i, where S signifies that it is the likelihood of being faulty and i indicates that line i is being investigated. For example, S_{125} represents the likelihood of line number 125 being a faulty line. A coefficient calculation is dependent on the method that is being used to calculate it, so for each method being analyzed, a different group of coefficients will be produced.

Program spectra are a collection of data that represents a specific view of the dynamic behavior of software (Abreu et al., 2007). In the context of fault localization, suppose we test a program of n lines (or units) by running m test cases. The hit spectra can be represented as an $m \times n$ matrix $M = (m_{ij})$, where $m_{ij} = 1$ if the run of test case i executes line j of the program, otherwise $m_{ij} = 0$. In addition, for each run of test case i, e_i is recorded as 1 (for failed test) or 0 (for successful test). Thus, the program spectra include information about each test run, whether the test is successful or not, and all the units of the program that were executed during the test. Most existing fault localizers use information from the hit spectra as described above to compute the similarity-coefficient. To do this, for each line (or unit) j of the program, we define:

$$a_{pq}(j) = \left| \{i \mid m_{ij} = p \wedge e_i = q\} \right|, where\ p, q \in \{0,1\}$$

Table 1 gives a summary of the descriptions for possible values of *p* and *q* for each line j.

Table 1 Descriptions of Similarity-Coefficient Elements for a Specific Line

Similarity-coefficient element	p	q	Description
$a_{00}(j)$	0	0	The number of test runs where line *j* was not executed and the test ended in success.
$a_{01}(j)$	0	1	The number of test runs where line *j* was not executed and the test ended in failure.
$a_{10}(j)$	1	0	The number of test runs where line *j* was executed and the test ended in success.
$a_{11}(j)$	1	1	The number of test runs where line *j* was executed and the test ended in failure.

These functions and definitions are used in defining the similarity metric used in the spectrum-based methods. Note that for line *j*, $a_{00}(j)$ is not of interest as it is not contributing to similarity factor.

Selected Existing Fault Localization Techniques

There are a large number of methods that have been proposed to solve the problem of automated fault localization. Tarantula (Jones et al., 2002) has been the method used in most comparisons and has shown to be the best when comparing both against spectrum-based fault localizers and other methods (Jones and Harrold, 2005). In recent studies, some methods have done better than Tarantula, most notably the Jaccard coefficient (Chen et al., 2002) and Ochiai coefficient (Meyer et al., 2005) in

the Abreu et al. study (Abreu et al., 2005). These three methods will be compared against the proposed method and are described in the following sections.

Tarantula

Tarantula was originally designed to provide a mechanism for visualizing faults within a program (Jones et al., 2002). The coefficient calculation that was developed to find the faulty line within a program was actually a formula for what color to display the line in a user interface, both by color and brightness. The color was chosen by

$$color = lowColor + \left(\frac{\%passed}{\%passed + \%failed}\right) * colorRange$$

where the low color was typically defined as red, with a range consisting of 120, which made the top of the color scale, the color green. The *%passed* and *%failed* is the percentage of tests that were successful and the percentage of tests that ended in failure in a given test set This would mean that lines that turned out with more of a red tint would be more likely to be faulty than yellow or green lines. For example, if a line was present in all of 100 successful tests and 1 of 10 failed tests, then the line would show light green, which would make it unlikely to be faulty.

The second component of the display is the brightness of a given line. The brightness represents how strong the results are, meaning that if a line appeared in a majority of success or failed test cases then it is more likely to be an accurate reading of whether the line is faulty. This is calculated by

$$brightness = \max(\%passed, \%failed)$$

which results in a value somewhere between zero and one. While this is an effective visualization tool, this was simplified for general fault localization into a formula for finding the coefficient for a set of lines (Jones and Harrold, 2005). This is represented by the given formula below.

$$S_i = \frac{\frac{a_{11}(i)}{(a_{01}(i) + a_{11}(i))}}{\left(\frac{a_{10}(i)}{a_{00}(i) + a_{10}(i)}\right) + \left(\frac{a_{11}}{a_{01}(i) + a_{11}(i)}\right)}$$

This is the formula that was used for comparison in our example. While Tarantula has been used as the method of comparison for many studies, the following two coefficients have been shown to be more effective in locating the correct faulty line.

Jaccard

The Jaccard similarity coefficient was most recently used in the area of large, dynamic web services or more generically, internet services, in the field of data clustering. This was used to watch where a request originated, follow the path it took through 1 to N servers, and try to locate not only where in the code chain that a request failed, but also locating which server the request failed on. This means it did not only have an interest in faults in the software, but also on the physical, or virtual, servers themselves.

The Jaccard coefficient has been used in explicit fault localization studies as well, where it was shown to be more effective than Tarantula, but less effective than the following method, the Ochiai coefficient (Abreu et al., 2006).

$$S_i = \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i) + a_{10}(i)}$$

The above formula will be using calculating the Jaccard coefficient in our study.

Ochiai

The Ochiai coefficient was originally designed for the molecular biology domain. This was used in the study of cluster analysis of vegetal species. The goal was to understand what the best grouping of individuals was to produce the best combination of characteristics. This was compared with a number of other coefficients, including the Jaccard coefficient in this domain (Meyer et al., 2004).

This method has also been shown to transfer well to the fault localization area of research. It is shown to currently be the best method of fault localization (Abreu et al., 2006). The formula that will be used in our comparison of this method is the following.

$$S_i = \frac{a_{11}(i)}{\sqrt{(a_{11}(i) + a_{01}(i)) * (a_{11}(i) + a_{10}(i))}}$$

CHAPTER IV

PROPOSED METHOD FOR SOFTWARE FAULT LOCALIZATION

The proposed method of automated fault localization builds on the foundation of program spectra and Dempster-Shafer theory. The following sections describe the mass functions used for fault localization, illustrate the process in a small example, and explain how it functions.

Mass Functions

The question being asked in fault localization is whether a given line is faulty without having to manually check. Using Dempster-Shafer theory, mass functions are the essential elements in estimating the likelihood of the faulty line or non-faulty line based on evidences from the program spectra described in Chapter III.

There are two possible hypotheses for any line *i* of program, namely it is faulty (FL_i) or it is not faulty ($\sim FL_i$). Thus, we have

$$U_i = \{FL_i, \sim FL_i\}$$

$$2^{U_i} = \{\emptyset, \{FL_i\}, \{\sim FL_i\}, \{FL_i, \sim FL_i\}\}$$

Two important results of each test run represented in the program spectra are whether the test was successful and what lines were executed during the test. There are two cases to be considered.

CASE1: failed test

If the test fails and the test involves the execution of the line in question, this should be evidence in support of this line being likely to be faulty. Thus, its likelihood of being non-faulty is zero. On the other hand, the likelihood of this line being faulty

can be estimated by a ratio of one over the total number of lines involved in this test run. We can formulate this formally as follows.

Let V_i be a binary variable representing execution of line *i*, where $V_i = 1$ if line *i* was executed, otherwise 0. The mass function for a failed test, m_f is defined for all possible subsets of the hypotheses in U_i as follows:

$$m_f(\sim FL_i) = 0.0$$

$$m_f(FL_i) = \alpha * \left(\frac{V_i}{\sum_j V_j}\right)$$

$$m_f(U_i) = 1 - m_f(FL_i)$$

The α in the equation represents the mass function weight, which will be discussed in more detailed later.

CASE2: successful test

If the test succeeds and the test involves the execution of the line in question, this should be evidence in support of this line being non-faulty. Thus, its likelihood of being faulty is zero. On the other hand, the likelihood of this line being non-faulty can be estimated by a ratio of one over the total number of lines involved in this test run. We can formulate this formally as follows.

The mass function for a success test, m_s, is defined for all possible subsets of the hypotheses in U_i as follows:

$$m_s(FL_i) = 0.0$$

$$m_s(\sim FL_i) = \beta * (\frac{V_i}{\sum_j V_j})$$

$$m_s(U_i) = 1 - m_s(\sim FL_i)$$

The β in the equation represents the mass function weight of the success test case. This will be discussed and applied in a later section.

The final step is to define the rule for combining mass functions. Dempster's rule of combination can be applied as shown below. For readability, we drop the subscript *i* that specifies line *i*.

$$m_1(FL) \oplus m_2(FL) = \frac{m_1(FL)m_2(FL) + m_1(FL)m_2(U) + m_1(U)m_2(FL)}{K}$$

$$m_1(\sim FL) \oplus m_2(\sim FL) = \frac{m_1(\sim FL)m_2(\sim FL) + m_1(\sim FL)m_2(U) + m_1(U)m_2(\sim FL)}{K}$$

$$K = 1 - \left(m_1(FL)m_2(\sim FL) + m_1(\sim FL)m_2(FL)\right)$$

$$m_1(U) \oplus m_2(U) = \frac{m_1(U)m_2(U)}{K}$$

Once these beliefs have been combined and new beliefs are generated, the new value of the conflict must be calculated for the next calculation series. This process can then be repeated until all tests have been executed and calculated. At that point, a final belief for each state being faulty is produced. The belief of being faulty becomes the similarity coefficient, S_i, for the proposed method, and therefore all lines are ranked based on this value.

Illustrated Example

In this section, an example program is given, analyzed, and the coefficients are generated for all four methods. The following is the sample program in Figure 1.

```
      mid() {
        int x, y, z, m;
1 :   read("Enter 3 numbers:", x, y, z);
2 :   m = z;
3 :   if (y < z)
4 :     if (x < y)
5 :       m = y;
6 :     else if (x < z)
7 :       m = y; /** bug **/
8 :   else
9 :     if (x > y)
10:       m = y;
11:     else if (x > z)
12:       m = x;
13:   print("Middle number is:", m);
      }
```

Figure 1 Source of Example Program

As shown in Figure 1, the program reads in three numbers and prints the middle number of the set. The fault is identified on line seven of the program, which causes the program to print the wrong number under specific circumstances. Suppose we apply six test cases. The tests are executed and the resulting program spectra are shown in Table 2.

Table 2 Example Program Spectra

Line *j*	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**
	5,3,4	**2,1,3**	**3,3,5**	**1,2,3**	**3,2,1**	**5,5,5**
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	1	1	1	1	0	0
5	0	0	0	1	0	0
6	1	1	1	0	0	0
7	0	1	1	0	0	0
8	0	0	0	0	1	1
9	0	0	0	0	1	1
10	0	0	0	0	1	0
11	0	0	0	0	0	1
12	0	0	0	0	0	0
13	1	1	1	1	1	1
Output	**4**	**1**	**3**	**2**	**2**	**5**
Expected	**4**	**2**	**3**	**2**	**2**	**5**
Result	**S**	**F**	**S**	**S**	**S**	**S**

In this table, the first row specifies the test number and second row specifies the input into the program. The rows labeled 1-13 specifies the line number and the values in the row indicate whether that line was executed during the given test, one meaning it was executed and zero meaning it was not. The final three rows indicate the actual output of the program with the specified input, the expected output, and whether the two outputs matched to produce a successful test (S) or they differed and the test failed (F).

In order to calculate the coefficient using the proposed method, the information must be retrieved from the table, input into the mass functions, and joined sequentially across all six tests to produce a calculated value. First, the mass function is selected based on the test result, so if the result ended in success (S), then the mass function m_s is used, otherwise m_f is used for a failed test (F). The values of *line j* are then used in the mass function to produce a set of beliefs for the given line *i*. The modified Dempster's Rule of Combination can then be used to join the beliefs across each test for a given line *i*. These coefficients were calculated along with the coefficients of all the comparing methods. For example, for the evidence of successful run of test1 of Table 2, applying the mass function m_s introduced in Chapter IV, we can obtain the mass function of line 6 resulting from this first evidence as follows:

$$m_1(FL_6) = 0$$

$$m_1(\sim FL_6) = \beta\frac{1}{6} = 0.0001 * 0.17 = 0.000017$$

$$m_1(U_6) = 1 - 0.000017 = 0.999983$$

These values represent the evidence after the first test. In this equation, the weight β is replaced, but the value used is discussed later. The second test resulted in failure, so

the mass function m_f is applied to line 6. The evidence gathered from this failed run of test 2 is:

$$m_2(FL_6) = \alpha\frac{1}{7} = 1.0 * 0.14 = 0.14$$

$$m_2(\sim FL_6) = 0$$

$$m_2(U_6) = 1 - 0.14 = 0.86$$

With evidence present for multiple runs, this can be combined into a single belief using the combination rule.

$$K_{1,2} = 1 - \big((0 * 0) + (0.000017 * 0.14)\big) = 0.99999762$$

$$m_{1,2}(FL_6) = \frac{(0 * 0.14) + (0 * 0.86) + (0.999983 * 0.14)}{K_{1,2}} \cong 0.1400$$

$$m_{1,2}(\sim FL_6) = \frac{(0.000017 * 0) + (0.000017 * 0.86) + (0.999983 * 0)}{K_{1,2}} \cong 0.00001$$

$$m_{1,2}(U_6) = \frac{0.999983 * 0.86}{K_{1,2}} \cong 0.8600$$

Continuing the execution with the execution of test 3, this test results in success, so m_s is used again to calculate the evidence.

$$m_3(FL_6) = 0$$

$$m_3(\sim FL_6) = \beta\frac{1}{7} = 0.0001 * 0.14 = 0.000014$$

$$m_3(U_6) = 1 - 0.000014 = 0.999986$$

With the generation of this new evidence, this can be combined with the previous evidence.

$$K_{1,2,3} = 1 - \big((0.14 * 0.000014) + (0.00001 * 0)\big) = 0.99999804$$

$$m_{1,2,3}(FL_6) = \frac{(0.14 * 0) + (0.14 * 0.999986) + (0.86 * 0)}{K_{1,2,3}} \cong 0.1400$$

$$m_{1,2,3}(\sim FL_6)$$

$$= \frac{(0.00001 * 0.000014) + (0.00001 * 0.999986) + (0.86 * 0.000014)}{K_{1,2,3}} \cong 0.00002$$

$$m_{1,2,3}(U_6) = \frac{0.86 * 0.999986}{K_{1,2,3}} \cong 0.8600$$

At this point, the similarity coefficient is represented by the belief of being fault (FL_6) with $S_6 = 0.14$. This process can then be repeated through all iterations of all lines to produce the final beliefs for the set of tests. The calculations are performed across all of the fault localization methods.

Table 3 Measures and Ranks Obtained from the Proposed Technique and Other Fault Localizers

Line *j*	Proposed [Rank]	Tarantula [Rank]	Jaccard [Rank]	Ochiai [Rank]
1	0.166654 [4]	0.500000 [4]	0.166667 [4]	0.408248[4]
2	0.166654 [6]	0.500000 [6]	0.166667 [6]	0.408248[6]
3	0.166654 [7]	0.500000 [7]	0.166667 [7]	0.408248[7]
4	0.166659 [3]	0.625000 [3]	0.250000 [3]	0.500000 [3]
5	0.00000 [10]	0.00000 [10]	0.00000 [10]	0.00000 [10]
6	0.166662 [2]	0.714286 [2]	0.333333 [2]	0.577350 [2]
7	0.166664 [1]	0.833333 [1]	0.500000 [1]	0.707107 [1]
8	0.00000 [11]	0.00000 [11]	0.00000 [11]	0.00000 [11]
9	0.00000 [12]	0.00000 [12]	0.00000 [12]	0.00000 [12]
10	0.00000 [8]	0.00000 [8]	0.00000 [8]	0.00000 [8]
11	0.00000 [9]	0.00000 [9]	0.00000 [9]	0.00000 [9]
12	0.00000 [13]	0.00000 [13]	0.00000 [13]	0.00000 [13]
13	0.166654 [5]	0.500000 [5]	0.166667 [5]	0.408248[5]

The line numbers with associated ranks and similarity coefficients for all four methods is shown in Table 3.This shows that all four methods were able to pinpoint the faulty line with line seven indicated as rank one. Also an important aspect in this example, line seven not only appeared in a failed test, but was also present in a

successful test. This means that all four methods have a tolerance for a faulty line being present in a successful test.

CHAPTER V

EVALUATION AND COMPARATIVE STUDY

This section describes how we empirically evaluate the proposed technique by experimenting with a benchmark obtained from real-world software that has become the de-facto standard for fault localization testing. This section also compares the results with other existing techniques. The section is organized by describing a data set obtained from the benchmark, how we obtain the program spectra for the proposed analysis for fault localization, evaluation metric, experimental setup and comparison results.

Benchmark Suite and Data Set

The Siemens program suite (Hutchins et al., 1994) is used as a benchmark for testing fault localizer effectiveness. The suite of seven programs has multiple versions which have had faults injected into them to mimic a real scenario. In the vast majority of the versions, this is a single fault, but there are some versions that have two faults in them. In these cases, the first fault found was used in calculations. Table 4 outlines the programs, the number of faulty versions of each program, the number of executable lines of code within the program, and the number of test cases available for the program. These programs were made available from the University of Nebraska and each program was accompanied by a predefined script to run all associated tests for the program. All programs are ANSI C code that uses standard input and output to communicate, so they have been designed to be piped to a file for later analysis.

There are 132 faulty versions present in the set, but only 119 were using in the experiment, where the following tests were excluded.

- Versions 4 and 6 of `print_tokens` due to having changes in the header file, not the source files
- Version 10 of `print_tokens2`, versions 3, 7, and 8 of `schedule`, version 9 of `schedule2`, and version 38 of `tcas` due to not presenting any failures

- Versions 5, 6, and 9 of `schedule` and versions 27 and 32 of `replace` due to crashing before GCov could output a trace file

Table 4 List of Siemens Programs

Program Name	# Versions	Lines of Code	# Test Cases
`print_tokens`	7	475	4130
`print_tokens2`	10	401	4115
`replace`	32	512	5542
`schedule`	9	292	2649
`schedule2`	10	297	2710
`tcas`	41	135	1608
`tot_info`	23	346	1052

Data Preprocessing and Acquisition of Program Spectra

There are a number of phases that occur to take the Siemens suite and produce results for comparison. The first phase is to execute each program and version to produce a set of trace information.

The environment used to execute the Siemens programs consisted of the following specifications.

- Microsoft Windows 7 Ultimate 64-bit
- GNU Compiler Collection (GCC) 4.4.0 compiler
- GNU Coverage (GCov) extension for GCC

The original scripts accompanied with the Siemens programs were designed to be run on UNIX and were modified accordingly to run identically on a Windows

environment. The GCC compiler was used to compile the programs for the use of GCov. GCov is designed to produce program coverage information when a program is executed. When a program is first compiled, it must be compiled with profiling information enabled. Once the program is compiled and executed, it will produce a GCov file. This file can then be read by GCov to produce individual source file coverage. It will outline which lines of code were executed and how many times it was executed. The original scripts were modified to add the production of this tracing information during the execution of each program version.

These programs, versions, and tests are used to generate the set of program spectra. With the faulty versions of each program, there is also a correct version, which produces all successful tests. When the correct version is executed across all tests, it produces a set of outputs for all the given inputs. This set of outputs can then be used to compare against the faulty version's outputs to infer whether the test was successful or not. For each faulty version, the program is executed against each input to produce an output. When all outputs have been generated, then the set of outputs from the correct version are compared against the set of outputs from the faulty version to gather which tests failed and which succeeded. This information is then bundled with the coverage information to understand what was executed in a given test and what the result of the test was.

This information is stored for analysis in a comma-separated-value (CSV) file that contains the whole of this information for each faulty version. It is given in the format in Table 5, where the line/input combination specified how many times the line was executed with the given input. The result of each input is 'true' if the test was successful and 'false' if it was not. This information constitutes the entirety of the program spectra used for calculating the similarity coefficients.

Table 5 Program Version CSV Results

Test	**Line1**	**Line2**	**…**	**LineN**	**Result**
Input1	< 0..n >	< 0..n >	…	< 0..n >	< T | F >
Input2	< 0..n >	< 0..n >	…	< 0..n >	< T | F >
…	…	…	…	…	…
inputN	< 0..n >	< 0..n >		< 0..n >	< T | F >

A CSV file with this content exists for each version of each program. This file is used as an input into the calculations to produce an output file that consists of the similarity coefficient calculations, ranked, with the associated line number it represents. The entire data generation process is summarized in a diagram shown in Figure 2.

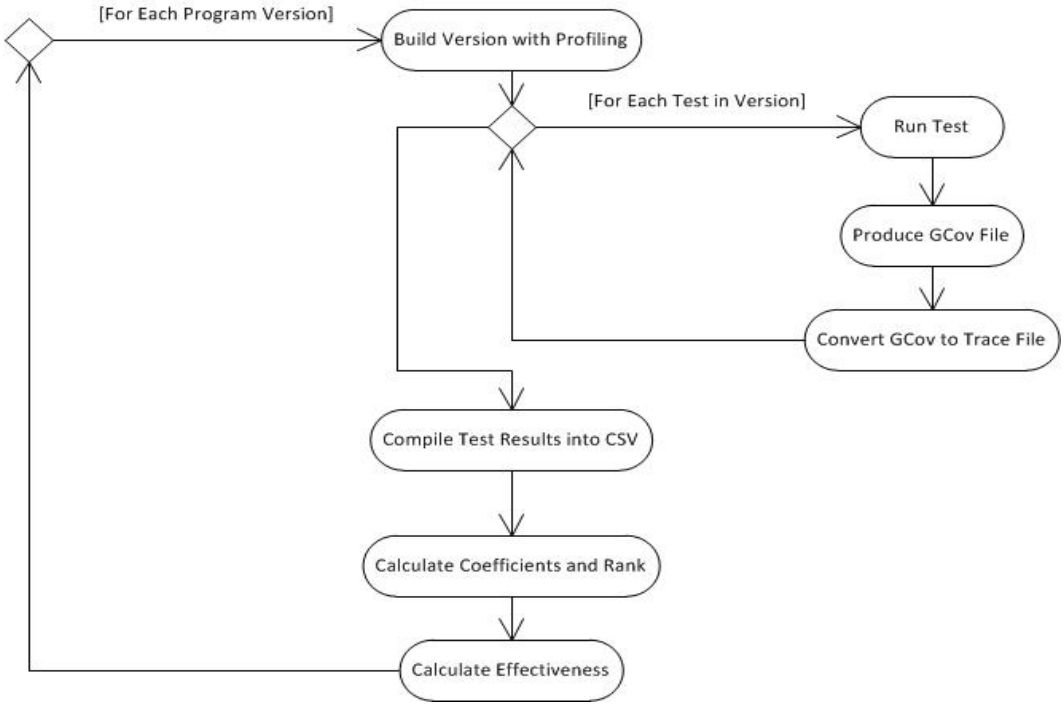

Figure 2 Version Execution Flow

With rankings and similarity coefficients produced for a given version, this information is used an input into a program that analyzes this file to produce the effectiveness measure for each method.

Evaluation Metric and Experimental Setup

Following a popular evaluation method introduced in (Jones et al., 2005), we use *effectiveness* as our primary evaluation metric. Since software fault localization is typically performed incrementally for one fault at a time, most existing spectrum-based localizers including ours assume that the program has a single fault. Effectiveness is then defined to be the percentage of code that did not need to be searched (saved efforts) to find the fault within a program. Suppose the software in consideration has N lines (or units). Let $R = \langle l_1, l_2, \ldots, l_N \rangle$ be a sequence of lines of the program in the order of their ranks (i.e., in descending order of similarity-coefficient values or mass function values). Let τ be the first line found in the sequence R such that it is the actual faulty line. In other words, the prediction of faulty line is ordered by its computed likelihood and so we take an optimistic view to see the very first line whose prediction matches with its actuality. Thus, τ and effectiveness can be defined below.

$$\tau = \{j \mid l_j in\ R\ is\ the\ faulty\ line\}$$

$$effectiveness = \left(1 - \left(\frac{\tau}{N}\right)\right) * 100\%$$

Although effectiveness has become the standard metric for fault localization, it has a problem when multiple lines have the same measure (i.e., coefficient value). In these cases, effectiveness becomes dependent solely on how they were ranked. The

best practice to this uses the *midline* adjustment (Ali et al., 2009). The midline takes the average rank of all the lines that have the same measure, which gives

$$\tau = \frac{k + m}{2}$$

$$k = \min_{1 \le j \le N}\{j \mid l_j \; in \; R \; has \; the \; same \; coefficient \; as \; the \; faulty \; line\}$$

$$k = \max_{1 \le j \le N}\{j \mid l_j \; in \; R \; has \; the \; same \; coefficient \; as \; the \; faulty \; line\}$$

Using these metrics, we evaluate the proposed approach and compare our results with the other three methods as will be discussed next.

Results

The four fault localization methods tested were all implemented in C++ and run side-by-side for producing calculations. The three comparative methods were all implemented directly in the test, but the proposed method was implemented in a separate static library that was included. The code that was used for the proposed method calculation is included in Appendix C.

The "optimistic" effectiveness and midline adjusted effectiveness were all calculated based on the generated similarity coefficients. The complete results obtained from the "optimistic" effectiveness are given in Appendix A and the results using the midline adjusted effectiveness are shown in Appendix B. In each of these tables, it displays the program name, version, actual faulty line number, total number of lines of executable code in the program, the rank of the faulty line under each approach, and the associated effectiveness.

In a total of 119 versions, there were no cases where the proposed method performed worse than any of the other methods and in 40% of all cases, it performed better than all three. This consisted of better results in 61 versions compared to

Tarantula, 59 versions compared to Jaccard, and 48 versions compared to Ochiai. This is shown on a version basis in Figure 3.

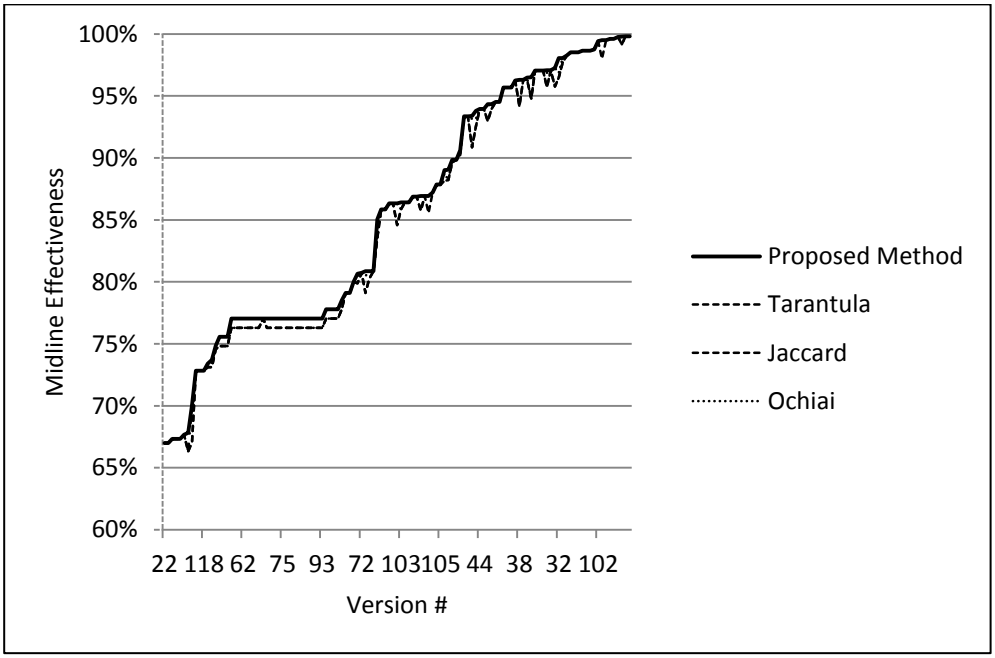

Figure 3 Midline Effectiveness by Version

The average effectiveness of each method is computed across all versions shown in Table 6. This includes the actual effectiveness average, midline effectiveness average, and the variance of both.

Table 6 Average Effectiveness and Variance Across All Versions

Method	**% Effective and Variance**		**Midline % Effectiveness and Variance**	
Proposed	88.43%	1.00%	85.57%	1.01%
Tarantula	88.15%	1.00%	85.09%	1.03%
Jaccard	88.17%	1.00%	85.11%	1.03%
Ochiai	88.36%	1.00%	85.28%	1.04%

This shows that all methods performed competitively with no more than 0.5% difference of the effectiveness and all performing well with over 88% average effectiveness. As shown in Table 6, the proposed method gives the highest average percentage of effectiveness, followed in order by Ochiai, Jaccard, and Tarantula. The proposed method also showed the least variance using midline, so it performed the most consistently.

Note that the weight factors in the mass functions can significantly impact the performance of our proposed method. When a line is present in a successful test, it is believed to be more likely of being a correct line, but the question is, how much more-so? When a test fails, it is a guarantee that at least one line is faulty within the execution, but that is not true for a successful test. In a successful test, there can still be one or more faulty lines present where the conditions are not met in order to express itself. This means that failed tests should be given more precedence than any other. This means that the weight for a failed test should be complete at 1.0 and the weight for a successful test should be set to the smallest amount possible. The values used in the study are shown in Table 7.

Table 7 Mass Function Weights

Variable	**Weight**
α	1.0000
β	0.0001

While these weights will cover the vast majority of applications, it is possible that the weight for a successful case, β, may need to be lowered in extremely large applications, with very large numbers of tests.

CHAPTER VI

CONCLUDING REMARKS AND FUTURE WORK

A method of software fault localization was introduced and compared to the top three methods of fault localization. The proposed method has shown to perform better than these methods in an empirical study. It also provides extensibility via mass functions. This is not present in any of the other methods and is due to the nature of Dempster-Shafer theory because one can always add a new mass function reflecting different type of evidence to contribute to the probability assignment for faulty hypotheses. The proposed technique is an on-line algorithmic approach that also adapts well to real-time systems, as it does not require any knowledge of previous executions, only the current beliefs.

There are areas of extension that may be the focus of future work. The first area is studying the observation quality and quantity metrics. The observation quality analyzes how a method behaves under varying types of test case results, where a faulty line may be present in either a successful or failed test case. The observation quantity analyzes how a method behaves under varying numbers of successful or failed test cases. The second area of research is in extending the proposed method to deal with multiple fault software.

BIBLIOGRAPHY

Abreu, Rui, Peter Zoeteweij and Arjan J.C. Gemund. "An Evaluation of Similarity Coefficients for Software Fault Localization." PRDC '06: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing. Washington: IEEE Computer Society, 2006. 39-46.

Abreu, Rui, Peter Zoeteweij and Arjan J.C. van Gemund. "On the Accuracy of Spectrum-based Fault Localization." TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. Washington: IEEE Computer Society, 2007. 89-98.

Ali, Shaimaa, et al. "Evaluating the Accuracy of Fault Localization Techniques." ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. Washington: IEEE Computer Society, 2009. 76-87.

Chen, M.Y., et al. "Pinpoint: problem determination in large, dynamic Internet services." Dependable Systems and Networks. 2002. 595-604.

Cleve, Holger and Andreas Zeller. "Locating causes of program failures." ICSE '05: Proceedings of the 27th international conference on Software engineering. St. Louis: ACM, 2005. 342-351.

Hutchins, Monica, et al. "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria." ICSE '94: Proceedings of the 16th international conference on Software engineering. Sorrento: IEEE Computer Society Press, 1994. 191-200.

Jones, James A. and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. Long Beach: ACM, 2005. 273-282.

Jones, James A., Mary Jean Harrold and John Stasko. "Visualization of test information to assist fault localization." ICSE '02: Proceedings of the 24th International Conference on Software Engineering. Orlando: ACM, 2002. 467-477.

Liblit, Ben, et al. "Scalable statistical bug isolation." PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. Chicago: ACM, 2005. 15-26.

Liu, Chao, et al. "Statistical debugging: A hypothesis testing-based approach." IEEE Transaction on Software Engineering. 2006, 831-848.

Meyer, Andreia da Silva, et al. "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize." Genetics and Molecular Biology. 2004, 83-91.

Nessa, S., et al. "Software Fault Localization Using N-gram Analysis." Wireless Algorithms, Systems, and Applications. 2008, 548-559.

Pearl, Judea. Causality: Models, Reasoning, and Inference. Cambridge University Press, 2000.

Popp, Karl. M. and Ralf Meyer. Profit from Software Ecosystems: Business Models, Ecosystems and Partnerships in the Software Industry. Norderstedt: BOD, 2010.

Renieres, M. and S.P. Reiss. "Fault localization with nearest neighbor queries." Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference. 2003. 30-39.

Reps, Thomas, et al. "The use of program profiling for software maintenance with applications to the year 2000 problem." ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering. Zurich: Springer-Verlag New York, Inc., 1997. 432-449.

Sedlmeyer, Robert L., William B. Thompson and Paul E. Johnson. "Diagnostic reasoning in software fault localization." IJCAI'83: Proceedings of the Eighth international joint conference on Artificial intelligence. Karlsruhe: Morgan Kaufmann Publishers Inc., 1983. 29-31.

Shafer, Glenn. A Mathematical Theory of Evidence. Princeton University Press, 1976.

Vessey, I. "Expertise in debugging computer programs: an analysis of the content of verbal protocols." IEEE Trans. Syst. Man Cybern. 1986, 621-637.

Weiser, Mark. "Programmers use slices when debugging." Commun. ACM. 1982, 446-452.

Wong, W. Eric and Yu Qi. "Bp Neural Network-Based Effective Fault Localization." International Journal of Software Engineering and Knowledge. 2009,: 573-597.

Wong, W. Eric, et al. "Using an RBF Neural Network to Locate Program Bugs." ISSRE. Seattle: IEEE Computer Society, 2008. 27-36.

Zeller, Andreas and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." IEEE Transactions on Software Engineering. 2002.

APPENDIX A

COMPARATIVE STUDY ACTUAL RANKS AND EFFECTIVENESS

Table A.1 Actual Rank and % Effective in print_tokens

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	225	46	90.53%	50	89.68%	50	89.68%	50	89.68%
2	225	25	94.95%	31	93.68%	31	93.68%	28	94.32%
3	233	1	100.00%	4	99.37%	4	99.37%	1	100.00%
5	251	5	99.16%	5	99.16%	5	99.16%	5	99.16%
7	279	29	94.11%	41	91.58%	41	91.58%	30	93.89%

Table A.2 Actual Rank and % Effective in print_tokens2

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	187	11	97.51%	18	95.76%	18	95.76%	18	95.76%
2	191	1	100.00%	7	98.50%	1	100.00%	1	100.00%
3	176	58	85.79%	64	84.29%	64	84.29%	64	84.29%
4	164	96	76.31%	108	73.32%	108	73.32%	96	76.31%
5	386	1	100.00%	1	100.00%	1	100.00%	1	100.00%
6	358	1	100.00%	1	100.00%	1	100.00%	1	100.00%
7	218	3	99.50%	3	99.50%	3	99.50%	3	99.50%
8	225	124	69.33%	130	67.83%	130	67.83%	130	67.83%
9	218	9	98.00%	15	96.51%	15	96.51%	9	98.00%

Table A.3 Actual Rank and % Effective in schedule

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	185	52	82.53%	52	82.53%	52	82.53%	52	82.53%
2	230	63	78.77%	63	78.77%	63	78.77%	63	78.77%
4	207	9	97.26%	9	97.26%	9	97.26%	9	97.26%

Table A.4 Actual Rank and % Effective in schedule2

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	135	20	93.60%	20	93.60%	20	93.60%	20	93.60%
2	295	119	60.27%	119	60.27%	119	60.27%	119	60.27%
3	292	116	61.28%	116	61.28%	116	61.28%	116	61.28%
4	92	18	94.28%	18	94.28%	18	94.28%	18	94.28%
5	110	90	70.03%	90	70.03%	90	70.03%	90	70.03%
6	77	10	96.97%	10	96.97%	10	96.97%	10	96.97%
7	292	117	60.94%	117	60.94%	117	60.94%	117	60.94%
8	275	109	63.64%	109	63.64%	109	63.64%	109	63.64%
10	28	70	76.77%	70	76.77%	70	76.77%	70	76.77%

Table A.5 Actual Rank and % Effective in replace

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	107	41	92.19%	42	91.99%	42	91.99%	41	92.19%
2	111	38	92.77%	40	92.38%	40	92.38%	39	92.58%
3	494	140	72.85%	149	71.09%	149	71.09%	142	72.46%
4	494	140	72.85%	143	72.27%	143	72.27%	140	72.85%
5	118	24	95.51%	31	94.14%	31	94.14%	24	95.51%
6	315	5	99.22%	13	97.66%	13	97.66%	11	98.05%
7	176	65	87.50%	65	87.50%	65	87.50%	65	87.50%
8	176	62	88.09%	68	86.91%	68	86.91%	62	88.09%
9	114	7	98.83%	7	98.83%	7	98.83%	7	98.83%
10	114	7	98.83%	7	98.83%	7	98.83%	7	98.83%
11	114	7	98.83%	7	98.83%	7	98.83%	7	98.83%
12	118	14	97.46%	25	95.31%	25	95.31%	14	97.46%
13	499	145	71.88%	148	71.29%	148	71.29%	145	71.88%
14	370	19	96.48%	26	95.12%	26	95.12%	20	96.29%
15	239	10	98.24%	12	97.85%	12	97.85%	10	98.24%
16	176	65	87.50%	65	87.50%	65	87.50%	65	87.50%
17	75	1	100.00%	1	100.00%	1	100.00%	1	100.00%
18	370	34	93.55%	34	93.55%	34	93.55%	34	93.55%
19	221	13	97.66%	13	97.66%	13	97.66%	13	97.66%
20	75	1	100.00%	1	100.00%	1	100.00%	1	100.00%
21	72	2	99.80%	2	99.80%	2	99.80%	2	99.80%
22	144	57	89.06%	57	89.06%	57	89.06%	57	89.06%
23	74	96	81.45%	96	81.45%	96	81.45%	96	81.45%
24	360	2	99.80%	2	99.80%	2	99.80%	2	99.80%

Table A.5 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
25	362	18	96.68%	19	96.48%	19	96.48%	18	96.68%
26	370	33	93.75%	35	93.36%	35	93.36%	34	93.55%
28	176	65	87.50%	74	85.74%	74	85.74%	65	87.50%
29	176	62	88.09%	62	88.09%	62	88.09%	62	88.09%
30	176	62	88.09%	69	86.72%	62	88.09%	62	88.09%
31	370	35	93.36%	35	93.36%	35	93.36%	35	93.36%

Table A.6 Actual Rank and % Effective in tcas

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	75	14	90.37%	14	90.37%	14	90.37%	14	90.37%
2	63	14	90.37%	14	90.37%	14	90.37%	14	90.37%
3	120	32	77.04%	32	77.04%	32	77.04%	32	77.04%
4	79	1	100.00%	1	100.00%	1	100.00%	1	100.00%
5	118	30	78.52%	30	78.52%	30	78.52%	30	78.52%
6	104	26	81.48%	26	81.48%	26	81.48%	26	81.48%
7	51	7	95.56%	7	95.56%	7	95.56%	7	95.56%
8	52	8	94.81%	8	94.81%	8	94.81%	8	94.81%
9	89	21	85.19%	21	85.19%	21	85.19%	21	85.19%
10	105	26	81.48%	26	81.48%	26	81.48%	26	81.48%
11	106	26	81.48%	26	81.48%	26	81.48%	26	81.48%
12	118	29	79.26%	29	79.26%	29	79.26%	29	79.26%
13	118	30	78.52%	30	78.52%	30	78.52%	30	78.52%

Table A.6 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
14	118	30	78.52%	30	78.52%	30	78.52%	30	78.52%
15	118	30	78.52%	30	78.52%	30	78.52%	30	78.52%
16	50	2	99.26%	2	99.26%	2	99.26%	2	99.26%
17	51	7	95.56%	7	95.56%	7	95.56%	7	95.56%
18	52	6	96.30%	6	96.30%	6	96.30%	6	96.30%
19	53	9	94.07%	9	94.07%	9	94.07%	9	94.07%
20	72	16	88.89%	16	88.89%	16	88.89%	16	88.89%
21	72	16	88.89%	16	88.89%	16	88.89%	16	88.89%
22	72	16	88.89%	16	88.89%	16	88.89%	16	88.89%
23	90	21	85.19%	21	85.19%	21	85.19%	21	85.19%
24	90	21	85.19%	21	85.19%	21	85.19%	21	85.19%
25	97	4	97.78%	4	97.78%	4	97.78%	4	97.78%
26	118	30	78.52%	30	78.52%	30	78.52%	30	78.52%
27	118	30	78.52%	30	78.52%	30	78.52%	30	78.52%
28	63	16	88.89%	16	88.89%	16	88.89%	16	88.89%
29	63	16	88.89%	16	88.89%	16	88.89%	16	88.89%
30	63	14	90.37%	14	90.37%	14	90.37%	14	90.37%
31	80	3	98.52%	3	98.52%	3	98.52%	3	98.52%
32	98	4	97.78%	4	97.78%	4	97.78%	4	97.78%
33	50	6	96.30%	6	96.30%	6	96.30%	6	96.30%
34	124	33	76.30%	33	76.30%	33	76.30%	33	76.30%
35	63	16	88.89%	16	88.89%	16	88.89%	16	88.89%
36	48	4	97.78%	4	97.78%	4	97.78%	4	97.78%
37	58	12	91.85%	12	91.85%	12	91.85%	12	91.85%

Table A.6 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
39	97	4	97.78%	4	97.78%	4	97.78%	4	97.78%
40	75	17	88.15%	17	88.15%	17	88.15%	17	88.15%
41	79	1	100.00%	1	100.00%	1	100.00%	1	100.00%

Table A.7 Actual Rank and % Effective in tot_info

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	343	25	93.06%	25	93.06%	25	93.06%	25	93.06%
2	85	4	99.13%	4	99.13%	4	99.13%	4	99.13%
3	77	27	92.49%	27	92.49%	27	92.49%	27	92.49%
4	233	22	93.93%	22	93.93%	22	93.93%	22	93.93%
5	105	23	93.64%	23	93.64%	23	93.64%	23	93.64%
6	71	1	100.00%	1	100.00%	1	100.00%	1	100.00%
7	378	59	83.24%	61	82.66%	61	82.66%	59	83.24%
8	201	20	94.51%	20	94.51%	20	94.51%	20	94.51%
9	106	24	93.35%	24	93.35%	24	93.35%	24	93.35%
10	372	58	83.53%	58	83.53%	58	83.53%	58	83.53%
11	198	10	97.40%	10	97.40%	10	97.40%	10	97.40%
12	177	37	89.60%	37	89.60%	37	89.60%	37	89.60%
13	394	67	80.92%	67	80.92%	67	80.92%	67	80.92%
14	75	83	76.30%	83	76.30%	83	76.30%	83	76.30%
15	200	11	97.11%	11	97.11%	11	97.11%	11	97.11%
16	99	85	75.72%	87	75.14%	87	75.14%	85	75.72%

Table A.7 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
17	223	16	95.66%	16	95.66%	16	95.66%	16	95.66%
18	308	91	73.99%	91	73.99%	91	73.99%	91	73.99%
19	55	75	78.61%	75	78.61%	75	78.61%	75	78.61%
20	308	37	89.60%	40	88.73%	40	88.73%	38	89.31%
21	75	81	76.88%	81	76.88%	81	76.88%	81	76.88%
22	352	112	67.92%	112	67.92%	112	67.92%	112	67.92%
23	215	11	97.11%	11	97.11%	11	97.11%	11	97.11%

APPENDIX B

COMPARATIVE STUDY MIDLINE RANKS AND EFFECTIVENESS

Table B.1 Midline Rank and % Effective in print_tokens

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	225	52	89.26%	56	88.42%	56	88.42%	56	88.42%
2	225	30	94.00%	36	92.74%	36	92.74%	33	93.37%
3	233	1	100.00%	4	99.37%	4	99.37%	1	100.00%
5	251	7	98.74%	7	98.74%	7	98.74%	7	98.74%
7	279	32	93.58%	44	91.05%	44	91.05%	33	93.37%

Table B.2 Midline Rank and % Effective in print_tokens2

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	187	14	96.76%	21	95.01%	21	95.01%	21	95.01%
2	191	2	99.75%	8	98.25%	2	99.75%	2	99.75%
3	176	60	85.29%	66	83.79%	66	83.79%	66	83.79%
4	164	120	70.32%	132	67.33%	132	67.33%	120	70.32%
5	386	1	100.00%	1	100.00%	1	100.00%	1	100.00%
6	358	2	99.75%	2	99.75%	2	99.75%	2	99.75%
7	218	5	99.00%	5	99.00%	5	99.00%	5	99.00%
8	225	129	68.08%	135	66.58%	135	66.58%	135	66.58%
9	218	11	97.51%	17	96.01%	17	96.01%	11	97.51%

Table B.3 Midline Rank and % Effective in schedule

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	185	61	79.45%	61	79.45%	61	79.45%	61	79.45%
2	230	61	79.45%	61	79.45%	61	79.45%	61	79.45%
4	207	11	96.58%	11	96.58%	11	96.58%	11	96.58%

Table B.4 Midline Rank and % Effective in schedule2

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	135	38	87.54%	38	87.54%	38	87.54%	38	87.54%
2	295	97	67.68%	97	67.68%	97	67.68%	97	67.68%
3	292	97	67.68%	97	67.68%	97	67.68%	97	67.68%
4	92	39	87.21%	39	87.21%	39	87.21%	39	87.21%
5	110	98	67.34%	98	67.34%	98	67.34%	98	67.34%
6	77	39	87.21%	39	87.21%	39	87.21%	39	87.21%
7	292	98	67.34%	98	67.34%	98	67.34%	98	67.34%
8	275	96	68.01%	96	68.01%	96	68.01%	96	68.01%
10	28	97	67.68%	97	67.68%	97	67.68%	97	67.68%

Table B.5 Midline Rank and % Effective in replace

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	107	52	90.04%	53	89.84%	53	89.84%	52	90.04%
2	111	48	90.82%	50	90.43%	50	90.43%	49	90.63%
3	494	98	81.05%	107	79.30%	107	79.30%	100	80.66%
4	494	98	81.05%	101	80.47%	101	80.47%	98	81.05%
5	118	29	94.53%	36	93.16%	36	93.16%	29	94.53%
6	315	10	98.24%	18	96.68%	18	96.68%	16	97.07%
7	176	70	86.52%	70	86.52%	70	86.52%	70	86.52%
8	176	67	87.11%	73	85.94%	73	85.94%	67	87.11%
9	114	7	98.83%	7	98.83%	7	98.83%	7	98.83%
10	114	7	98.83%	7	98.83%	7	98.83%	7	98.83%
11	114	7	98.83%	7	98.83%	7	98.83%	7	98.83%
12	118	19	96.48%	30	94.34%	30	94.34%	19	96.48%
13	499	99	80.86%	103	80.08%	103	80.08%	100	80.66%
14	370	15	97.27%	22	95.90%	22	95.90%	16	97.07%
15	239	10	98.24%	12	97.85%	12	97.85%	10	98.24%
16	176	70	86.52%	70	86.52%	70	86.52%	70	86.52%
17	75	1	100.00%	1	100.00%	1	100.00%	1	100.00%
18	370	31	94.14%	31	94.14%	31	94.14%	31	94.14%
19	221	15	97.27%	15	97.27%	15	97.27%	15	97.27%
20	75	1	100.00%	1	100.00%	1	100.00%	1	100.00%
21	72	2	99.80%	2	99.80%	2	99.80%	2	99.80%
22	144	52	90.04%	52	90.04%	52	90.04%	52	90.04%
23	74	98	81.05%	98	81.05%	98	81.05%	98	81.05%
24	360	2	99.80%	2	99.80%	2	99.80%	2	99.80%

Table B.5 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
25	362	18	96.68%	19	96.48%	19	96.48%	18	96.68%
26	370	29	94.53%	31	94.14%	31	94.14%	30	94.34%
28	176	70	86.52%	79	84.77%	79	84.77%	70	86.52%
29	176	67	87.11%	67	87.11%	67	87.11%	67	87.11%
30	176	67	87.11%	74	85.74%	67	87.11%	67	87.11%
31	370	31	94.14%	31	94.14%	31	94.14%	31	94.14%

Table B.6 Midline Rank and % Effective in tcas

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	75	27	80.74%	27	80.74%	27	80.74%	27	80.74%
2	63	31	77.78%	32	77.04%	32	77.04%	32	77.04%
3	120	31	77.78%	32	77.04%	32	77.04%	32	77.04%
4	79	2	99.26%	2	99.26%	2	99.26%	2	99.26%
5	118	31	77.78%	32	77.04%	32	77.04%	32	77.04%
6	104	31	77.78%	32	77.04%	32	77.04%	32	77.04%
7	51	31	77.78%	32	77.04%	32	77.04%	32	77.04%
8	52	31	77.78%	32	77.04%	32	77.04%	32	77.04%
9	89	31	77.78%	32	77.04%	32	77.04%	32	77.04%
10	105	31	77.78%	32	77.04%	32	77.04%	32	77.04%
11	106	31	77.78%	31	77.78%	31	77.78%	31	77.78%
12	118	30	78.52%	31	77.78%	31	77.78%	31	77.78%
13	118	31	77.78%	32	77.04%	32	77.04%	32	77.04%

Table B.6 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
14	118	31	77.78%	32	77.04%	32	77.04%	32	77.04%
15	118	31	77.78%	32	77.04%	32	77.04%	32	77.04%
16	50	26	81.48%	26	81.48%	26	81.48%	26	81.48%
17	51	31	77.78%	32	77.04%	32	77.04%	32	77.04%
18	52	29	79.26%	30	78.52%	30	78.52%	30	78.52%
19	53	31	77.78%	32	77.04%	32	77.04%	32	77.04%
20	72	31	77.78%	32	77.04%	32	77.04%	32	77.04%
21	72	31	77.78%	32	77.04%	32	77.04%	32	77.04%
22	72	31	77.78%	32	77.04%	32	77.04%	32	77.04%
23	90	31	77.78%	32	77.04%	32	77.04%	32	77.04%
24	90	31	77.78%	32	77.04%	32	77.04%	32	77.04%
25	97	4	97.78%	4	97.78%	4	97.78%	4	97.78%
26	118	31	77.78%	32	77.04%	32	77.04%	32	77.04%
27	118	31	77.78%	32	77.04%	32	77.04%	32	77.04%
28	63	33	76.30%	34	75.56%	34	75.56%	34	75.56%
29	63	33	76.30%	34	75.56%	34	75.56%	34	75.56%
30	63	31	77.78%	32	77.04%	32	77.04%	32	77.04%
31	80	4	97.78%	4	97.78%	4	97.78%	4	97.78%
32	98	5	97.04%	5	97.04%	5	97.04%	5	97.04%
33	50	31	77.78%	32	77.04%	32	77.04%	32	77.04%
34	124	30	78.52%	31	77.78%	31	77.78%	31	77.78%
35	63	33	76.30%	34	75.56%	34	75.56%	34	75.56%
36	48	30	78.52%	31	77.78%	31	77.78%	31	77.78%
37	58	31	77.78%	32	77.04%	32	77.04%	32	77.04%

Table B.6 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
39	97	4	97.78%	4	97.78%	4	97.78%	4	97.78%
40	75	30	78.52%	31	77.78%	31	77.78%	31	77.78%
41	79	2	99.26%	2	99.26%	2	99.26%	2	99.26%

Table B.7 Midline Rank and % Effective in tot_info

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
1	343	23	93.64%	23	93.64%	23	93.64%	23	93.64%
2	85	6	98.55%	6	98.55%	6	98.55%	6	98.55%
3	77	49	86.13%	49	86.13%	49	86.13%	49	86.13%
4	233	15	95.95%	15	95.95%	15	95.95%	15	95.95%
5	105	42	88.15%	42	88.15%	42	88.15%	42	88.15%
6	71	2	99.71%	2	99.71%	2	99.71%	2	99.71%
7	378	47	86.71%	49	86.13%	49	86.13%	49	86.13%
8	201	23	93.64%	23	93.64%	23	93.64%	23	93.64%
9	106	42	88.15%	42	88.15%	42	88.15%	42	88.15%
10	372	49	86.13%	49	86.13%	49	86.13%	49	86.13%
11	198	19	94.80%	19	94.80%	19	94.80%	19	94.80%
12	177	47	86.71%	47	86.71%	47	86.71%	47	86.71%
13	394	47	86.71%	47	86.71%	47	86.71%	47	86.71%
14	75	94	73.12%	94	73.12%	94	73.12%	94	73.12%
15	200	19	94.80%	19	94.80%	19	94.80%	19	94.80%
16	99	91	73.99%	93	73.41%	93	73.41%	92	73.70%

Table B.7 Continued

Version	Faulty Line #	Proposed Rank and %		Tarantula Rank and %		Jaccard Rank and %		Ochiai Rank and %	
17	223	15	95.95%	15	95.95%	15	95.95%	15	95.95%
18	308	87	75.14%	88	74.86%	88	74.86%	88	74.86%
19	55	94	73.12%	94	73.12%	94	73.12%	94	73.12%
20	308	38	89.31%	41	88.44%	41	88.44%	39	89.02%
21	75	92	73.70%	93	73.41%	93	73.41%	93	73.41%
22	352	94	73.12%	94	73.12%	94	73.12%	94	73.12%
23	215	15	95.95%	15	95.95%	15	95.95%	15	95.95%

APPENDIX C

PROPOSED METHOD LIBRARY CALCULATION CODE

FaultyTuple.h

```
#ifndef _FAULTYTUPLE_H_
#define _FAULTYTUPLE_H_

#include <string>
using namespace std;

/**
@author ALJ
@date 2010-06-27
@file faultytuple.h
@version 1.0
This class defines a tuple for calculating and
maintaining belief.
*/
class FaultyTuple
{
public:
    /**
    Default constructor.
    */
    FaultyTuple();

    /**
    Constructor- initializes the tuple.
    @param[in] name The name of the element.
    */
    FaultyTuple(wstring name);

    /**
    Copy constructor- duplicates a tuple.
    @param[in] The object to copy from.
    */
    FaultyTuple(const FaultyTuple& other);

    /**
    Equal operator overload.
    @param[in] The object to copy from.
    @return This.
    */
    FaultyTuple& operator=(const FaultyTuple&
        other);

    /**
    Virtual destructor.
    */
    virtual ~FaultyTuple();

    /**
    Get the name of the element.
    @return The name.
    */
    wstring GetName() const;

    /**
    Get the current belief of being normal.
```

```
    @return The current normal belief.
    */
    double GetNormalBelief() const;

    /**
    Get the current belief of being faulty.
    @return The current fault belief.
    */
    double GetFaultyBelief() const;

    /**
    Get the current belief of being suspect.
    @return The current conflict belief.
    */
    double GetSuspectBelief() const;

    /**
    Add an execution of this element.
    @param[in] execSequence The sequence number
        in which this element was executed
    from 0 to n - 1.
    @param[in] execTotal The total number of
        elements executed in the sequence, n.
    @param[in] execSuccessful Whether the
        execution ended in success, otherwise
        failure.
    */
    void AddExecution(size_t execSequence,
size_t execTotal, bool execSuccessful);

protected:
    /**
    Add and recalculate the updated beliefs
        based on the added masses.
    @param[in] normal The normal belief to add.
    @param[in] faulty The faulty belief to add.
    @param[in] suspect The conflict belieft to
        add.
    */
    void AddMassValues(double normal, double
        faulty, double suspect);

private:
    bool m_isInitialized;
    double m_normal;
    double m_faulty;
    double m_suspect;
    wstring m_name;
    const double m_weight_baseSuccess;
    const double m_weight_baseFailure;
};

#endif
```

FaultyTestSet.h

```
#ifndef _FAULTYTESTSET_H_
#define _FAULTYTESTSET_H_

#include <string>
#include <list>
#include <map>
#include "faultytuple.h"
using namespace std;

/**
@file faultytestset.h
@author ALJ
@date 2010-06-28
@version 1.0
This class encapsulates the functionality of
calculating and building a ranking
of elements that may be faulty.
*/
class FaultyTestSet
{
public:
    /**
    Default constructor.
    */
    FaultyTestSet();

    /**
    Virtual destructor.
    */
    virtual ~FaultyTestSet();

    /**
    Add an execution sequence to the
        calculation.
    @param[in] execList The list of executed
        statements.
    @param[in] wasSuccessful Whether the
        execution ended in success.
    */
    void AddExecution(const list<wstring>&
        execList, bool wasSuccessful);

    /**
    Get the faulty tuple for the specified
        index.
    @param[in] index The index of the tuple.
    @return The faulty tuple at the specified
        index.
    */
    const FaultyTuple* GetFaultyTuple(size_t
        index) const;

    /**
    Get the total number of faulty tuples.
    @return The total number of tuples
    */
    size_t GetTotalFaultyTuples() const;

private:
    map<wstring, FaultyTuple> m_tuples;
};

#endif
```

FaultyTuple.cpp

```
#include "faultytuple.h"

FaultyTuple::FaultyTuple()
    : m_weight_baseSuccess(0.0001),
    m_weight_baseFailure(1.00)
{
    this->m_name = L"";
    this->m_isInitialized = false;
    this->m_normal = 0.0;
    this->m_faulty = 0.0;
    this->m_suspect = 0.0;
}

FaultyTuple::FaultyTuple(wstring name)
    : m_weight_baseSuccess(0.0001),
    m_weight_baseFailure(1.00)
{
    this->m_name = name;
    this->m_isInitialized = false;
    this->m_normal = 0.0;
    this->m_faulty = 0.0;
    this->m_suspect = 0.0;
}

FaultyTuple::FaultyTuple(const FaultyTuple&
other)
    : m_weight_baseSuccess(0.0001),
    m_weight_baseFailure(1.00)
{
    this->operator=(other);
}

FaultyTuple::~FaultyTuple()
{
}

FaultyTuple& FaultyTuple::operator=(const
        FaultyTuple& other)
{
    this->m_name = other.m_name;
    this->m_isInitialized =
        other.m_isInitialized;
    this->m_normal = other.m_normal;
    this->m_faulty = other.m_faulty;
    this->m_suspect = other.m_suspect;
    return *this;
}

wstring FaultyTuple::GetName() const
{
    return this->m_name;
}

double FaultyTuple::GetNormalBelief() const
{
    return this->m_normal;
}

double FaultyTuple::GetFaultyBelief() const
{
    return this->m_faulty;
}

double FaultyTuple::GetSuspectBelief() const
{
    return this->m_suspect;
}
```

```
void FaultyTuple::AddExecution(size_t
        execSequence, size_t execTotal, bool
        execSuccessful)
{
      /* execSequeunce is a placeholder for
      future ability to build mass functions
      based on execution sequence */

    if (execSequence >= execTotal)
    {
        return;
    }

    double dExecTotal = (double)execTotal;

    // if the execution ended in success
    if (execSuccessful)
    {
        // add the base success mass function
        double normal = this->
                m_weight_baseSuccess * (1.0f /
                dExecTotal);
        double faulty = 0.0f;
        double conflict = 1.0f - normal;
        this->AddMassValues(normal, faulty,
                conflict);
    }
    // else it ended in failure
    else
    {
        // add the base failure mass function
        double normal = 0.0f;
        double faulty = this->
                m_weight_baseFailure * (1.0f /
                dExecTotal);
        double conflict = 1.0f - faulty;
        this->AddMassValues(normal, faulty,
                conflict);
    }
}

void FaultyTuple::AddMassValues(double normal,
        double faulty, double suspect)
{
    // check if the initial values have been
        set
    if (!this->m_isInitialized)
    {
        this->m_normal = normal;
        this->m_faulty = faulty;
        this->m_suspect = suspect;
        this->m_isInitialized = true;
    }
    else
    {
        // get the measure of conflict between
                the two masses
        double k = 1 - ((this->m_normal *
                faulty) + (this->m_faulty *
                normal));
        // get the updated normal belief
        this->m_normal = ((this->m_normal *
                normal) + (this->m_normal *
                suspect) + (normal * this->
                m_suspect)) / k;
        // get the updated faulty belief
        this->m_faulty = ((this->m_faulty *
                faulty) + (this->m_faulty *
                suspect) + (faulty * this->
                m_suspect)) / k;
```

```
            // calculate the updated belief of
                    suspect
            this->m_suspect = (this->m_suspect *
                    suspect) / k;
        }
    }
```

FaultyTestSet.cpp

```
#include "faultytestset.h"

FaultyTestSet::FaultyTestSet()
{
    this->m_tuples.clear();
}

FaultyTestSet::~FaultyTestSet()
{
}

void FaultyTestSet::AddExecution(const
        list<wstring>& execList, bool
        wasSuccessful)
{
    map<wstring, int> combinedExecs;
    int sequence = 0;

    for (list<wstring>::const_iterator it =
        execList.begin() ; it != execList.end()
        ; it++)
    {
        if (combinedExecs.find(*it) ==
                combinedExecs.end())
        {
            combinedExecs[*it] = sequence;
            sequence++;
        }
    }
    for (map<wstring, int>::iterator it =
        combinedExecs.begin() ; it !=
        combinedExecs.end() ; it++)
    {
        if (this->m_tuples.find(it->first) ==
                this->m_tuples.end())
        {
            this->m_tuples[it->first] =
                FaultyTuple(it->first);
        }
        this->m_tuples[it->
                first].AddExecution(it->second,
                sequence, wasSuccessful);
    }
}

const FaultyTuple*
FaultyTestSet::GetFaultyTuple(size_t index)
const
{
    map<wstring, FaultyTuple>::const_iterator
        it = this->m_tuples.begin();

    for (size_t i = 0 ; i < index ; i++)
    {
        it++;
    }

    return &(it->second);
}

size_t FaultyTestSet::GetTotalFaultyTuples()
        const
{
    return this->m_tuples.size();
}
```