Technical Report CS15-02

An Eclipse-based Integrated and Automated Fault Localization System

Tristan Challener

Submitted to the Faculty of
The Department of Computer Science

Project Director: Dr. Gregory Kapfhammer
Second Reader: Dr. John Wenskovitch

Allegheny College
2015

I hereby recognize and pledge to fulfill my responsibilities as defined in the Honor Code, and to maintain the integrity of both myself and the college community as a whole.

Tristan Challener

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The process of debugging can be complex and difficult. The first task associated with debugging is identifying the fault, and this step, *fault localization*, is the most expensive in terms of time cost [4]. In this chapter, we discuss current techniques for fault localization. In addition, we highlight the major goals of this research project.

1.1 Current State of the Art

At present, the process of fault localization is mostly manual, though several techniques exist that attempt to automate the process. A number of these techniques are described in detail below. Many of these automated fault localization techniques (AFLs) compare the flow of execution through a faulty program when executing passed and failed test cases. This type of approach employs *coverage monitoring*, the tracking of which statements are executed by test cases. Any statement that is executed is said to be *covered*. Tools such as Java Code Coverage (JaCoCo) [3] provide information regarding which statements are covered by a supplied test suite. By analyzing code coverage for different test cases, these AFLs seek to determine which statements are most likely to contain the fault.

Many Coverage tools exist, including those designed for integrated development environments such as Eclipse. For example, JaCoCo provides interactive coverage monitoring within Eclipse. However, JaCoCo only reports total coverage for a test suite; that is, the tool reports which statements were covered, partially covered, or not covered after execution of all of the test cases. This system does not report on coverage of individual test cases. Since *per-test coverage* is vital to the application of several fault localization techniques, the lack of such information leads to increased difficulty in applying those techniques.

As an alternative to the more popular JaCoCo, we will make use of a lesser known tool called CodeCover. Though a number of interfaces are provided, including command-line and Apache Ant, we will focus on the Eclipse plugin for the purpose of this project. Unlike JaCoCo, CodeCover is designed to produce coverage on a per-test basis. The system includes support for breaking coverage information for a JUnit test suite across test methods, automatically generating distinct coverage data for each test method. This functionality makes CodeCover ideal to the goals of this project.

1.2 Goals of the Project

There are two significant goals for this project. The first is the empirical evaluation of existing risk evaluation functions to determine the most effective on average. This is a necessary step in the process toward completion of our second goal, because no study has made the specific comparison necessary.

The second goal of this project is the development of an Eclipse plugin which displays the per-test coverage from a test suite and per-statement suspiciousness analysis, through an interactive system, as well as to demonstrate its effectiveness. The plugin will allow a programmer to select individual or multiple test cases and view their coverage, or to select individual statements and view a list of covering test cases as well as suspiciousness rating and ranking.

We hypothesize that making per-test coverage (in addition to suspiciousness information), the basis for many fault localization techniques, readily available will allow programmers to more rapidly identify faults in programs. This hypothesis will be investigated through human study, employing experienced programmers, specifically undergraduate students, to compare the benefits of per-test coverage when compared to full suite coverage. To generate test cases for evaluating our plugin, we will introduce faults into programs using the MAJOR mutation analysis system [6].

1.3 Thesis Outline

This thesis consists or five chapters, which cover a wide variety of topics related to the project. In Chapter 2, we discuss past approaches to problems in the area of fault localization. In addition, we cover existing tools which we will incorporate into our final system or make use of to produce intermediate data. Chapter 3 features a thorough discussion of the method of approach used to achieve the results. Specifically, we relate the details of our test environment and setup for the empirical comparison of risk evaluation functions, including test programs, method of producing per-test coverage, and system for evaluating risk evaluation functions. In addition, Chapter 3 covers implementation details of the Eclipse plugin.

Chapter 4 provides the empirical results of the risk evaluation function comparison study. This includes various data visualization figures and a discussion of these data. We also discuss our conclusions as a result of this study, as well as the impact these results have on our plugin implementation. Chapter 5 discusses the human study used to evaluate our plugin, beginning with the format and setup details and including the results of the study. We also discuss our conclusions based on the results of the study. The final chapter reviews the project as whole, a annotates difficulties encountered, reiterates conclusions drawn in the previous chapters, and a discusses future work.

Chapter 2

Related Work

2.1 Automated Fault Localization Techniques

There are a wide variety of existing fault localization techniques [4]. Many of these methods make use of per-test coverage analysis, such as Set-union, Set-intersection, Nearest Neighbor, and Tarantula. There are other AFLs that do not use per-test coverage, such as Cause Transitions [1], but they are outside the scope of this project.

Set-union and Set-intersection compare the coverage of a single failed test case to the coverage of all of the passed test cases. Set-union defines the initial set of suspicious statements by the set of all test cases visited by the failed test case but not visited by any passed test cases. Conversely, Set-intersection defines the initial set as the set of all statements visited by every passed case but not by the failed test case. Nearest Neighbor works similarly to the previous techniques, but instead of considering all of the passed test cases, it only considers one. By some method, the passed test cases that is most similar to the failed test case is identified. Nearest Neighbor defines the initial set as the set of all statements visited by the failed case and not by the passed case.

Tarantula is notably different from the previously mentioned techniques. Instead of determining a set of suspicious statements, Tarantula ranks techniques in order of suspiciousness. However, Tarantula still utilizes per-test coverage reporting. In their paper, Jones and Harrold present an empirical analysis which indicates the superiority of Tarantula in both efficiency and effectiveness when compared to the other techniques described above, as shown in Figure 1 [4]. Since Tarantula uses per-test coverage, this provides substantial evidence for the importance of per-test coverage in the fault localization process.

Xie et al. present an alternative discussion of various risk evaluation functions focusing primarily on theoretical correctness [13]. They argue that many existing risk evaluation formulas are actually functionally equivalent. In addition, they provide theoretical analysis that shows the superiority of specific functions relative to one another both within functionally equivalent groups and between those groups. Through detailed theoretical proof they show that the equivalent sets ER1 and ER5 are maximal for accurately calculating suspiciousness. The ER1 set includes a pair

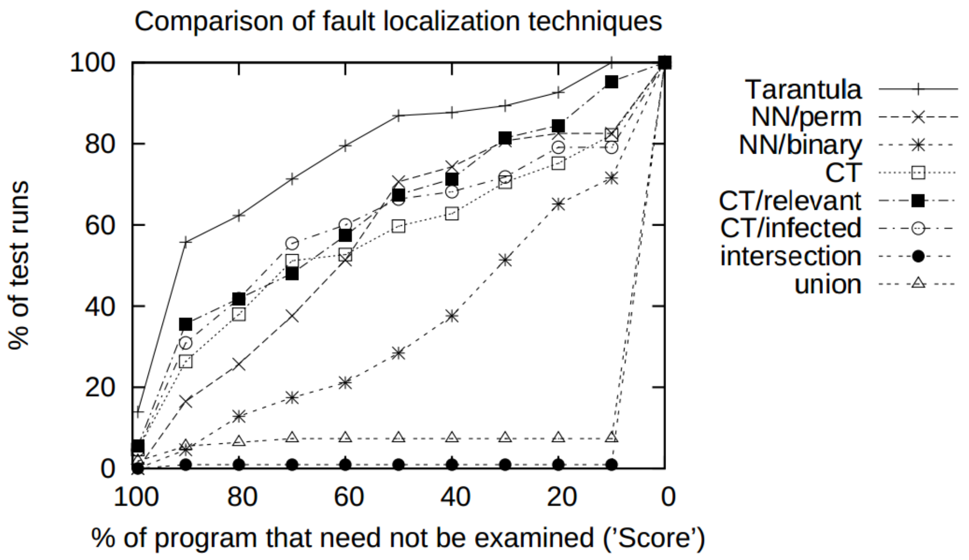


Figure 2.1: An empirical comparison of several fault localization techniques by percentage of code eliminated from consideration. The graph plots percentage eliminated (determined by the rank of the faulty statement) against frequency that the tool achieved that level of success. Tarantula is demonstrated to achieve better results more often than other techniques.(from [4])

of formulas studied in a paper by Naish et al. [7] The ER5 equivalent set includes three functions, two of which are also studied in the paper by Naish et al. The third is discussed in a paper by Wong et al. [12] It is important to note that all the above risk evaluations also require per-test coverage data.

As a followup to the paper by Xie et al. mentioned above, Qi et al. [9] performed a detailed empirical to study the practical effectiveness of the theoretically maximal functions as proven by Xi et al. This study includes a wide variety of risk evaluation functions, and compares them by applying them to practical, real-world programs and faults. The various formulas were inserting them into popular automated program repair system GenProg. GenProg uses genetic programming, applying a risk evaluation function to repeatedly alter a faulty program by mutating suspicious statements. In their study, Qi et al. modified GenProg to use several different formulas. By comparing the effectiveness of automated repair when using each different risk evaluation function, they were able to provide substantial empirical evidence regarding the superiority of certain formulas. Note that this method of comparison differs from that pursued by Jones and Harrold [4], discussed previously, which compared the relative suspiciousness ranking of faulty statements (referred to as EXAM score).

Surprisingly, Qi et al. did not confirm the results shown by Xie et al. Instead, they found that several functions, eliminated from consideration as maximal functions early in the proof provided by Xie et al., were actually better than expected. Specifically, Qi et al. show that in practice, the Jaccard risk evaluation function performed equally as well or better than all other formulas considered. They conclude that, at least within the context of automated program repair, Jaccard should be favored when choosing a risk evaluation function. In addition, they further conclude that the EXAM score method of comparing these formulas does not accurately reflect their relative performance in the context of automated program repair.

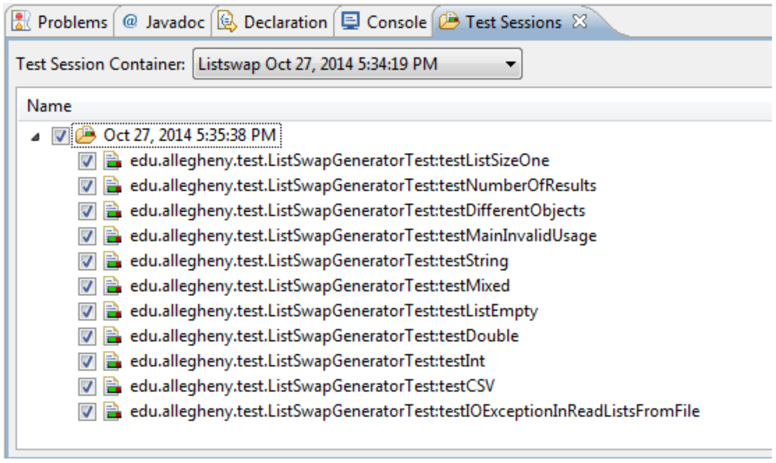
In another paper [8], Parnin and Orso performed a human study about the real-world effectiveness of fault localization techniques. In that study, they asked participants, graduate students in computer science with varying levels of experience, to perform debugging tasks. Participants were asked to compare their experiences when using standard debugging practices with using an Eclipse plugin designed to display Tarantula suspiciousness ranking. This plugin was as simple as possible, to eliminate from consideration any other factors besides suspiciousness ranking. The only functionality provided was a list of suspicious statements which, when selected, navigated the environment to that line in the source code. When evaluating their hypotheses, he authors considered the relative average amount of time required for each task, with and without the plugin. They also considered the relative experience level of the participants, which they evaluated based on number of tasks completed. In addition, they also experimented with artificially raising and lowering the rank of the fault statement in the suspicious statement list. Finally, Parnin and Orso used their plugin to record the order in which users visited the suspicious statements, and asked the participants to complete a questionnaire about their experiences and use of the tool.

Parnin and Orso came to a number of conclusions regarding several different comparisons. First, they noticed that in general, the participants did not on average complete the tasks faster with statistical significance. However, they did notice a significant improvement in performance when using the tool if the subject in question was more experience. This suggests that, if the tool were more refined and participants more used to the tool, it might actually be significantly more helpful. They also found evidence that suggests that the exact rank of the statement does not significantly affect the difficulty of locating the fault when using the tool. In particular, participants often navigated the list of statements in a non-linear fashion: sometimes they skipped ahead or backwards. When skipping ahead, the authors suggest that participants may have been skipping over code blocks they had already deemed unlikely to contain the bug. Finally, they conclude that there is no such thing as perfect bug understanding in practice. That is, simply seeing a faulty statement out of context is not sufficient to identify and repair the bug.

We notice first that a large number of the negative results from Parnin and Orso's study, in regards to the effectiveness of automatic fault localization, may have stemmed from the simplicity of the Eclipse plugin tested. We therefore hypothesize that with a more sophisticated tool, programmers will have significantly improved performance over use of traditional debugging techniques. Since the goal of our tool is to provide per-test coverage as context for suspiciousness, we suggest that our tool is likely to outperform traditional debugging.

2.2 CodeCover Coverage Analysis

CodeCover produces coverage information for Java, and has built in functionality for generating coverage information on a per-test suite, per-test case, and even per-test method basis. In addition, CodeCover is designed to function within Eclipse. All parts of the coverage monitoring process can be done through the Eclipse development environment. Since the final goal of this project is to produce an integrated Eclipse tool for fault localization, using per test coverage and risk evaluation, CodeCover is ideally suited to our purposes.



Test Session Container: Listswap Oct 27, 2014 5:34:19 PM

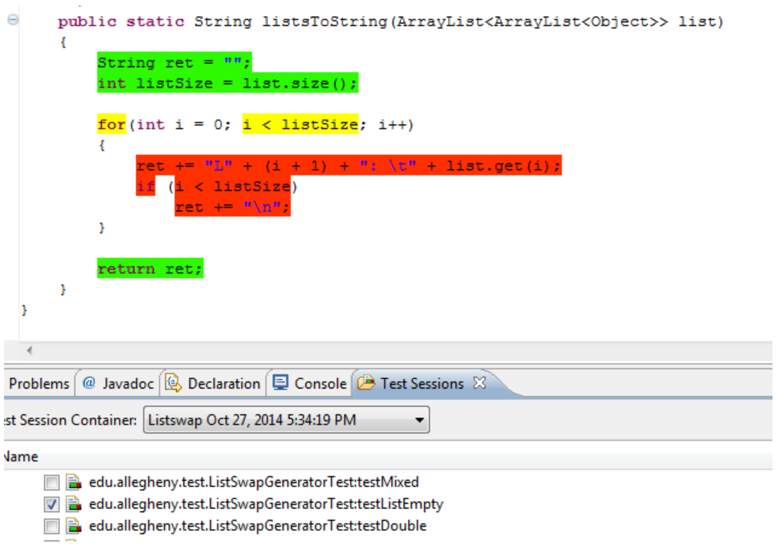
```
Oct 27, 2014 5:35:38 PM
  edu.allegheny.test.ListSwapGeneratorTest:testListSizeOne
  edu.allegheny.test.ListSwapGeneratorTest:testNumberOfResults
  edu.allegheny.test.ListSwapGeneratorTest:testDifferentObjects
  edu.allegheny.test.ListSwapGeneratorTest:testMainInvalidUsage
  edu.allegheny.test.ListSwapGeneratorTest:testString
  edu.allegheny.test.ListSwapGeneratorTest:testMixed
  edu.allegheny.test.ListSwapGeneratorTest:testListEmpty
  edu.allegheny.test.ListSwapGeneratorTest:testDouble
  edu.allegheny.test.ListSwapGeneratorTest:testInt
  edu.allegheny.test.ListSwapGeneratorTest:testCSV
  edu.allegheny.test.ListSwapGeneratorTest:testIOExceptionInReadListsFromFile
```

Figure 2.2: Screenshot of CodeCover Eclipse plugin displaying per-test coverage output from a single execution of a JUnit test suite.

CodeCover works in several stages. The first step to acquiring coverage information is instrumentation. During instrumentation, numerous statements are procedurally inserted into the Java source files which are to be included in the coverage analysis. In the context of Eclipse, these files are manually marked, since there may be source files for which coverage information is unnecessary. The statements that are inserted during instrumentation record various information as the code is executed, including which existing Java statements were executed, and which test case or method exeucted them. After instrumentation is complete, the newly instrumented source files, as well as any other source files, are compiled as normal.

CodeCover provides functionality for using existing JUnit test cases and test suites for coverage analysis. With an existing Eclipse project with a JUnit test class, CodeCover can simply be enabled and executed. By default, CodeCover produces output in the form of a several coverage reports. The output includes a full coverage report for every test method within the JUnit test; that is, per-test coverage data, as shown in Figure 2.2.

After executing a test process through the CodeCover Eclipse, per-test coverage can be viewed in several ways. First, coverage can be viewed on a per-test basis within Eclipse. As shown in Figure 2.3, selecting one or more test methods displays coverage



```
    public static String listsToString(ArrayList<ArrayList<Object>> list)
    {
        String ret = "";
        int listSize = list.size();

        for(int i = 0; i < listSize; i++)
        {
            ret += "L" + (i + 1) + ": \t" + list.get(i);
            if (i < listSize)
                ret += "\n";
        }

        return ret;
    }
}
```

```
st Session Container: Listswap Oct 27, 2014 5:34:19 PM
Name
[ ] edu.allegheny.test.ListSwapGeneratorTest:testMixed
[✓] edu.allegheny.test.ListSwapGeneratorTest:testListEmpty
[ ] edu.allegheny.test.ListSwapGeneratorTest:testDouble
```

Figure 2.3: Screenshot of CodeCover Eclipse plugin displaying coverage highlighting for a single test method. Statements highlighted in green were covered by this test method, and those in red were not.

information for those test methods only. In addition to viewing coverage within Eclipse, CodeCover allows coverage information to be executed in a variety of formats. Using provided template XML (eXtensible Markup Language) files, coverage data can be exported, for example, as hierarchical HTML (HyperText Markup Language) documents or as CSV (Comma Separated Value) files.

2.3 MAJOR Mutation

MAJOR is a fault seeding and mutation analysis tool that integrates directly into the Java compiler [6]. *Mutation* is the process of intentionally introducing faults into a program for various purposes, including evaluation of test suite quality and testing for fault localization techniques. Mutants generated are either *killed* by the test suite or *live*. A mutant is killed if a test case fails; that is, the test suite was sufficient to identify the fault. Live mutants either indicate that the test suite is insufficient or that the introduced fault was in fact logically equivalent to the original code.

MAJOR can perform a number of conditional mutations, including replacing binary arithmetic, logical, relational, shift and unary operators with valid alternatives. Figure 4 shows an example of one possible mutation performed by MAJOR. The tool can also replace literal values with alternatives; these various mutation types can be selected with compiler operators. Along with the domain specific language provided by MAJOR, this allows for great flexibility and extensibility. Additionally, MAJOR has been shown to be relatively efficient, with only about 15% runtime overhead. It is also noteworthy that MAJOR utilizes coverage analysis to improve efficiency.

A recent paper by Just et al. [5] indicates a statistical correlation between mutant

```
public static Type classify(int a, int b, int c) {
    int trian;
    if (a <= 0 || b <= 0 || c <= 0)
      return Type.INVALID;
    trian = 0;
          ... (additional statements) ...
}
```

```
public static Type classify(int a, int b, int c) {
    int trian;
    if ((a <= 0 || b <= 0) != c <= 0)
      return Type.INVALID;
    trian = 0;
          ... (additional statements) ...
}
```

Figure 2.4: (top) Original program and (bottom) program after mutation.

detection and real world fault detection. Although they also note that as much as 20% of real world faults cannot be represented by mutants, this still suggests that mutants are a viable substitute for real world faults in the context of software testing and, similarly, fault localization.

Chapter 3

Method of Approach

In this chapter, we discuss the implementation details for our system. The implementation can be broken into four distinct sections:

1. XML parsing
2. Intermediate representation
3. Simplified representation
4. Risk evaluation

3.1 XML Parsing

When CodeCover produces per-test coverage information, it stores the results in a container. Though CodeCover features several coverage report export types, including hierarchical HTML, these reports do not include all information necessary for this project. Specifically, these export formals only include general per-test coverage information, displaying percent coverage for each test. They do not include information related to precisely which tests executed each statement, which is a requirement for suspiciousness analysis.

Due to the limitations in CodeCover export formats, we are forced to make use of the more complex XML container. Since the container is in XML format, we must first parse the file before we can perform analysis. To that end, we utilize DOM (document object model) parsing to store the entire container as a tree structure. We chose DOM over SAX(simple API for XML) due to the complexity of the container file. Since SAX does not store the document in memory in its entirety, multi-pass analysis is both more efficient and less complex when using DOM parsing.

3.2 Intermediate Representation

After DOM parsing is complete, provided no errors occurred, the document tree root is passed into the intermediate representation constructor. The IR (intermediate representation) for the container is designed to be simple to build while traversing a DOM

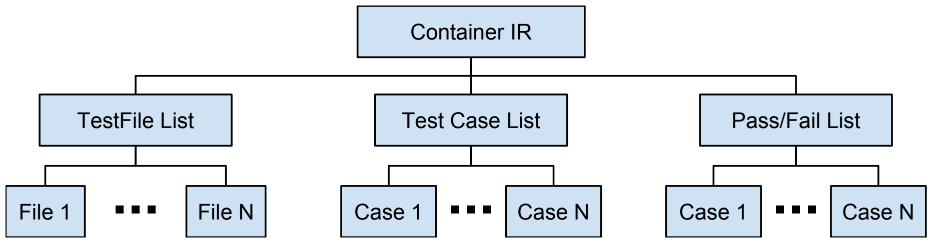


Figure 3.1: Diagram of data structure for IR.

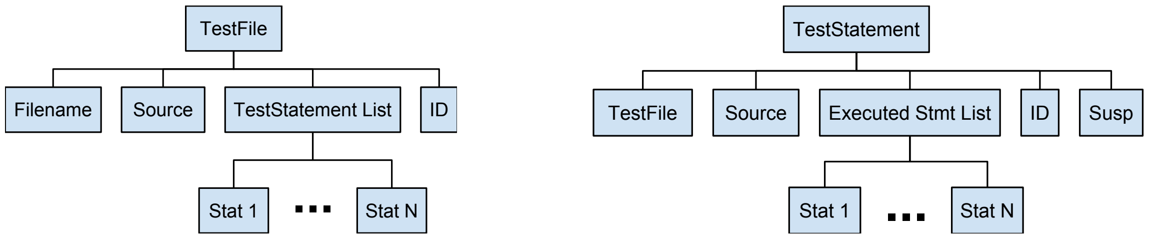


Figure 3.2: Diagram of data structures used within the IR to store files (left) and statements (right)

of a CodeCover container. The `ContainerIR` object contains a list of `TestFile`s, as well as a list of `String` names of test cases. The object also includes a boolean values, in which the i^{th} element indicates the pass/fail status of the i^{th} test case. A visualization of the container IR, as well as its internal data structures, can be found in Figures 3.1 and 3.2 respectively.

Each `TestFile` object contains the name and source code of the file as a `String`, the internal identification numberfor the file as an `int`, and a list of `TestStatement`s. A `TestStatement` includes an ID and source code for the statement, each stored as `String`s. In addition, the `TestStatement` object stores a `TestFile` parent object and the suspiciousness value associated with the statement. Finally, the statement includes a `boolean` list, where each element corresponds to a test case. These boolean values indicate whether the respective test case executed this statment. A `TestStatement` can be uniquely identified by the combination of its source file and ID.

CodeCover coverage containers include all required information, divided into three sections. First, the container lists all source files included in coverage analysis, including the entirety of their source code, the name of the file, and a unique internal identification value. Following the source files is a hierarchical list of all statements within those files. These statements are defined by by their category, source file, and character offset within the source file. For this project, we only consider basic statements (in Java, this essentially refers to executable statements terminated with semicolons).

The final section of a CodeCover container is the per-test coverage data itself. This is represented by a series of test cases, each of which contains a child for every file executed by that test. Under each file is a list of specific statements executed by that test inside that file. These statements are identified by their alphanumeric identifier.

Our system processes these sections in the order they appear above. Using the DOM produced from parsing, we can easily traverse the relevant segments of the container. Traversal of the DOM tree consists of repeated iterations of sub-levels of the XML hierarchy, in order to locate a node with a specific name. Throughout this process, we make no assumptions about the sequence in which nodes appear. Rather than assume that, for example, a given node is always the first child, we always use a loop structure to make certain we identify the correct node.

In order to identify all source files defined by the XML container, we first locate the `SrcFileList` node. We then iterate through its children to locate all `SrcFile` nodes. For each `SrcFile` node found, we extract the attributes of that node and add a `TestFile` to the `ContainerIR` file list.

Once we have located the source files, we must next identify the statements. To do so, we first locate the hierarchical statement definition root. Beginning with that root node, we recursively traverse the entire sub-tree. For each `BasicStmnt` node, we add a new statement to the correct `TestFile`. In order to do so, we must first identify the `LocList` child node, followed by its `Loc` child node. The attributes of the `Loc` node include the source code character offset for the basic statement in question. Between the attributes of the `BasicStmnt` node and those of the `Loc` node, we extract the information necessary to create a new `TestStatement` object. Once this recursive traversal of the statement hierarchy is complete, the `TestFile`s in the `ContainerIR` will contain all basic statements.

Since we now have a complete list of basic statements, we can process the coverage data section of the container to identify test cases and their coverage. To begin, we first iteratively locate the `TestSession` node, whose children include a `TestCase` node for each test case in the test suite. We add the name of each test case to the `ContainerIR` test case list, then process the children of the node to find coverage data. Each test case node contains a hierarchy of `CovList`, `CovPrefix`, and `Cov` nodes. We locate all `CovPrefix` nodes, which each correspond to a single file, then retrieve the `TestFile` corresponding to this node from the list of test files within the `ContainerIR`. Finally, we iterate through all `Cov` children of the `CovPrefix` to find all basic statements covered by a given test case within a given file. For each statement identified, we mark the corresponding `TestStatement` as covered.

In addition to coverage data, a `TestCase` node also contains a `Comment` attribute. This content of this attribute is comprised of the entire text output produced when the test case was executed. When the test case passes, this attribute is empty; however, when a failure occurs, the description of the failure is stored in this node. As a result, we can observe this attribute to determine the pass/fail status of a given test. We ascertain this data by checking the `Comment` attribute for the starting string `"Failure"`. Since this word always heads the description of a failure, the `Comment` of a failed test will always begin with the same text. This pass/fail value is recorded in the boolean list stored in the `ContainerIR`, where true indicates a passed test and false indicates a failed one.

3.3 Simplified Representation

The intermediate representation discussed in section 3.2 is appropriate as a preliminary data structure. However, it is not conducive to suspiciousness analysis. All of the risk evaluation functions considered in this project require four pieces of information for each statement that must be evaluated: $ae_f, ae_p, an_f,$ and an_p [13]. These variables represent the number of test cases that meet certain requirements for a given statement. Variables ae_f and ae_p denote the number of test cases that executed the given statement and failed or passed respectively. Similarly, an_f and an_p refer to the number of test cases that did not execute the statement and passed or failed, respectively. These four values become the input to each of the suspiciousness functions. Therefore, the simplified representation that best suits our needs is one that stores these values for each statement, with no extraneous information.

Before converting the intermediate representation to the simplified form described above, we must first address the issue of pass/fail status for test cases. We resolve this problem by programmatically executing the JUnit test suite, with its class name provided at run time, that tests the system evaluated by CodeCover. We achieve this by first using Java Reflection to acquire the `Class` object associated with the class name provided. This `Class` then becomes the input to `JUnitCore.run()`. The `JUnitCore` class executes the specified `Class` object, parsing the suite to locate all tests. The `run()` method executes the test class, returning a `Result` object.

A `Result` object provides a list of `Failure` objects, which each includes the fully qualified name of the test case that generated the error. By extracting only the method name portion of this full name, we can identify which test methods resulted in failures. Since CodeCover also by default uses test method names to label test cases, we can compare these values directly to determine which of the test cases in the IR failed. This provides the final necessary component for suspiciousness analysis.

In order to convert the IR to the described simple representation, the first step is build a list of `TestInfo` objects, which represent individual test cases. Each `TestInfo` object includes a pass/fail boolean value, as well as a boolean list that describes, for each statement under test, whether that statement was executed by this test. We extract this information from the IR by iterating through the list of test case names. For each test case, we check the execution status of that test case for each statement, storing this information in a list (where each element corresponds to a single statement, and true indicates the statement was executed by this test). Next, we compare the name of the test case in question to the list of failures provided by the `Result` object discussed previously. If the test case appears in the list of failures, we mark the test as failing.

Iterating through the `TestInfo` list, we can determine the $ae_f, ae_p, an_f,$ and an_p of each statement. For each test case, we check whether the test passed then determine which statements it executed by iterating through the boolean list stored

```
public class Jaccard implements REFunction {
  public double analyze( int aef, int aep, int anf, int anp ) {
    return ( (double) aef / (aef + anf + aep ) );
  }

  public String toString() {
    return "Jaccard";
  }
}
```

Figure 3.3: Implementation of REFunction interface for Jaccard function.

in the `TestInfo` object. For each statement, we increment the appropriate variable depending on its pass/fail and execution status. The result is a list, for each of the four variables, in which each element corresponds to its value for the corresponding statement. After completing this process, we have a `CoverageReport` object with all of the information necessary for risk evaluation.

3.4 Risk Evaluation

There are many different risk evaluation functions, but they are structurally similar. We internally represent risk evaluation functions using our `REFunction` interface. This interface provides abstract methods for returning the name of the function and for evaluating the function on provided $ae_f, ae_p, an_f,$ and an_p. Figure 3.3 provides an example implementation of the REFunction interface. For each function included, we create an implementation of `REFunction`. A complete list of `REFunctions` is stored in a static array of the same name in `CoverageReport` for ease of iteratively performing all evaluation functions.

When performing risk evaluation, we iterate through the specified functions. For each of these functions, we store the results of analysis in a `ResultsList` data structure. This structure allows for ease of sorting analyzed statements by suspiciousness rating. In addition, the suspiciousness values of the `ResultsList` are normalized to a zero-to-one scale according to

$$L_i = \frac{L_i - L_{min}}{L_{max} - L_{min}}.$$

In this equation, L_i refers to the i^{th} element of the `ResultsList`, and L_{max}, L_{min} refer to the maximum and minimum values in the list, respectively. Though normalizing the results does not change the ordering of the statements, it does make the actual suspiciousness values easier to compare across multiple different functions. The final result of analysis is a list of statements, ordered from most suspicious to least suspicious, with values ranging from zero (not suspicious) to one (very suspicious).

3.5 Overview

When executing the system, the main class, `Test.java`, must be invoked with three additional parameters. The first of these is the fully qualified name of the JUnit test class (e.g. `edu.allegheny.listswap.ListSwapGeneratorTest`), which will be programmatically executed to obtain pass/fail information. This test class *must* be in the classpath, or the system will not be able to execute the tests. The second parameter is the fully qualified file name for the CodeCover container to be analyzed. The final parameter is the name for the output file—the output path is fixed to the (to be created) results directory within the present working directory.

Execution begins with parsing the XML container. When the document has been stored as a DOM, it is passed forward to the `ContainerIR` constructor, which processes the document into an IR as described previously. The IR is then passed to the `CoverageReport` class, where the test class is executed and coverage information is collated into the simple representation. Suspiciousness evaluation is then completed using all risk evaluation functions specified.

After performing suspiciousness evaluation, the results are written to an output file in CSV (comma separated value) format. The attributes *Function, Statement ID, Filename, Suspiciousness, Rank, Statement Count,* and *Case Application* serve to uniquely identify a statement while providing information on risk evaluation. This format conforms to *tidy data* [11], a "standard way of mapping the meaning of a dataset to its structure", in that each variable forms a column and each observation forms a row. This organizational structure allows for reliably simple evaluation of data using statistical analysis tools. Though tidy data contains extensive redundancy, that redundancy simplifies the data analysis and visualization process.

Chapter 4

Experimental Results

Appendix A

Addendum

In this addendum we discuss in further detail the human study, including details regarding number of participation, pre-evaluation of participants, and study setup. In addition, we discuss our contingency plan if we encounter unexpected difficulty. The purpose of this addendum is to address concerns related to feasibility of the project and lack of exact details for the evaluation strategy.

A.1 Human Study

The evaluation strategy for the Eclipse plugin that will be the final product of this project will be evaluated using a human study. This human study will involve twenty undergraduate student participants. with varying experience in the areas of coverage analysis, automated testing, and development with Eclipse (or similar integrated development environment). The participants will be drawn from the Allegheny computer science department majors and minors, and will not accept those students who have not completed or are in the process of completing at least two computer science courses.

Since undergraduate students will have widely varying experience in the areas of coverage analysis, automated testing, and development with Eclipse (or similar integrated development environment), we will ask participants to self-evaluate these attributes of their computer science experience in order to allow us to better balance the groups in the study. This evaluation will take the form of simple scale values; each question will ask to the participant to rate his experience in that area on a scale from one to ten, with one representing complete unfamiliarity and ten representing high confidence. We will then divide the twenty participants into two groups with roughly even levels of experience. When evaluating overall level of experience, we will consider. In addition, we will ask students to list courses completed and to specify their industry experience (including jobs or internships) so that we may incorporate that information into our final results. This questionnaire will be provided to students well in advance of the actual study, so that we can make decisions on grouping before the start of the study.

Once students have been divided into groups, each group will be assigned two faulty programs to debug (the students will work independently). For each of the two tasks, students will be allotted no more than 30 minutes to complete the debugging process. The purpose of this limitation is to avoid pushing the participants to exhaustion and thus affecting the results. Prior to beginning the tasks, we will instruct the students on the use of our plugin. We will allow 30 minutes for this instruction period, to ensure that participants are at least conversant in the use of the tool. This will eliminate the possibility that the tool was not helpful only because the students were not familiar with its functions.

We will use the same two programs, with the same fault, for each group of students. However, the first group of students will use our plugin to debug the first program, while they will use traditional debugging techniques to repair the fault in the second program. In contrast, the second group of participants will use our plugin for the second program and traditional debugging for the first program. This organization of tasks allows us to compare average completion times for each task when using or not using our plugin. Notice that this experiment setup is derived from that used by Parnin and Orso for their research paper in a similar area [8]. That study included 24 graduate students, two programs, and a similar division of students into groups; that is, where each group used or did not use the fault localization tool for opposite tasks, just as we have described.

After completion of the two tasks, or after each time limit has expired, we will ask students to complete a survey related to their experience when using the tool. Questions asked will include the following:

- In what ways did you find the plugin to be helpful for locating the fault?
- In what ways do you feel the plugin could be improved?
- Please briefly describe your debugging process when using the tool and when not using the tool.
- Did you find the tool to cause an overall benefit or detriment to your debugging process?
- In what situations would you consider using this tool again?
- In what situations would you prefer to use traditional debugging techniques?

Participants' answers to subjective question such as these will allow us to better gauge the strengths and weaknesses of the plugin, as well as evaluate the effectiveness of the tool from the perspective of the users. These answers will be particularly helpful if we find that the plugin does not result in a statistically significant increase in debugging performance, because it will allow us to determine in what ways the plugin could be improved. By considering these answers, we can further evaluate the reason for negative results. We can identify whether automatic fault localization is actually unhelpful, or if improvements to the plugin could give more positive results. The questions will also be helpful in the event of positive results, because we can identify which features of the tool are most effective: per-test coverage, suspiciousness ranking, or absolute suspiciousness value.

A.2 Contingency Plan

Because this project is highly ambitious, especially considering the time constraints, any unforeseen difficulties could result in our inability to complete all of our goals. We will establish a contingency plan in the event that we discover at some point in the project that we will not be able to meet the deadline.

The first portion of the project deals with parsing CodeCover per-test coverage data from XML into a simplified intermediate representation, so that we may perform risk evaluation. This component will be very complex, and is in itself an ambitious project. It involves first designing a system to parse the highly complex XML test session containers produced by CodeCover, which will have to be tested for correctness. Then we must select a large number of test programs, which must have existing JUnit test suites for per-test coverage analysis. We will then need to import these programs into Eclipse, perform initial CodeCover setup, and execute the coverage monitoring tool on the existing test cases. We will then have to use or parsing system to extract the relevant data for every program tested and perform risk evaluation for every function we decide to study. Finally, we must correlate all of the resulting data inside R to produce effective visualizations, as well as analyze and discuss these results in writing.

Since any portion of this component of the project may prove to be too complex for the given time constraints, our contingency plan will involve completion of only this part of the project. We will leave development and evaluation of the Eclipse plugin as future work, and will discuss the difficulties that led to our use of this contingency. Since we will not have completed the Eclipse plugin, the implementation details will not be included in Chapter 3. In addition, we will omit Chapter 5 entirely, since its only purpose is discussion of the method and results of the human study. The revised thesis outline and project deadlines will be as follows:

Task	**Completion Deadline**
Chapters 1 and 2	December 15
Simplified Representation	February 3
Empirical Study	February 28
R Data Analysis	March 10
Chapter 4	March 24
Chapter 5	April 3
Oral Defense	April 6 - 24
Final Thesis	May 1

Table A.1: Proposed work schedule for contingency plan

1. Introduction
2. Related Work
3. Method of Approach
4. Empirical Results
5. Conclusion

Note that we will only follow this new schedule and outline if we determine that it will not be possible to achieve the original deadlines. The current plan is still to pursue the original schedule as defined in our project proposal. This is only a contingency plan, and we will not utilize it unless necessary. Regardless of difficulties, however, we will at least complete the first goal of the project: empirical comparison of risk evaluation functions (as described in Section 1.2).

Bibliography

[1] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[2] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[3] Mountainminds GmbH, Co. KG, and Contributors. Eclemma - java code coverage for eclipse, 2014.

[4] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[5] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[6] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011.

[8] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[9] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 191–201, New York, NY, USA, 2013. ACM.

[10] Walter F. Tichy. Hints for reviewing empirical work in software engineering. *Empirical Softw. Engg.*, 5(4):309–312, December 2000.

[11] Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10):??–??, 9 2014.

[12] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188 – 208, 2010. Computer Software and Applications.

[13] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31:1–31:40, October 2013.