BadPair: A Framework for Automated Software Testing

by

Chien-Hsing Chang
B.Sc., University of Victoria, Victoria, B.C., Canada, 1996
B.B.A., Simon Fraser University, Burnaby, B.C., Canada, 2006

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

BadPair: A Framework for Automated Software Testing

by

Chien-Hsing Chang
B.Sc., University of Victoria, Victoria, B.C., Canada, 1996
B.B.A., Simon Fraser University, Burnaby, B.C., Canada, 2006

Supervisory Committee

Dr. Daniel M. Hoffman, Supervisor
(Department of Computer Science)

Dr. Daniel German, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Daniel M. Hoffman, Supervisor
(Department of Computer Science)

Dr. Daniel German, Departmental Member
(Department of Computer Science)

Abstract

Testing every possible combination of the input parameter values is often impractical, inefficient or too expensive. One common alternative is pairwise testing where every pairwise combination of the parameter values is tested. Although pairwise testing significantly reduces the number of test cases, the challenge remains in analyzing the test outputs to discern the precise characteristics of parameters causing the failures. This thesis proposes a novel approach to output analysis by identifying "bad pairs": pairs that always result in failed test cases. A framework implementing the proposed approach is presented together with three case studies. Results from the case studies suggest there are positive relationships among the numbers of failed test cases, faults, and independent bad pairs. Also, filtering of test cases seems to have a significant impact on the bad pairs identified. We believe the proposed approach can facilitate the debugging process in software testing.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my greatest gratitude to Professor Daniel Hoffman, my supervisor. Throughout my study at the University of Victoria for the Master of Science in Computer Science, Professor Hoffman has been providing with me enormous encouragement and support in many aspects. The thesis would not been complete without his superior guidance. Additionally, I would like to thank my beloved wife for her unconditional support.

Chapter 1
Introduction

Software testing is costly: it is both time-consuming and labour-intensive. By some measurements, at least 20% of overall software development costs arise from software testing [4, 8]. But uncorrected faults in software can be even costlier. Software errors cost the U.S. economy nearly $60 billion dollars [23]. On its own, testing is not enough to guarantee that the system-under-test (SUT) is free of bugs. As Dijkstra put it, "Program testing can be used to show the presence of bugs, but never their absence" [13]. Nonetheless, it adds confidence in the correctness of the SUT.

Failures of software resulting from combinations of input parameters have long been studied and researched [16]. In the telephony industry, the problems of interaction among various parameters are well known and have been vigorously researched [15]. Exhaustive testing where every possible combination of the parameter values is tested on the SUT is often impractical and too expensive to execute [5, 18]. The sheer volume of combinations of multiple parameter values alone presents a challenging task in the debugging process in software testing. Instead, one remedy is to test the SUT on a subset of all possible combinations as various studies suggest that it can be an effective or practical option [5, 7]. However, the number of combinations to test on the SUT can remain huge when the number of parameters and the number of parameter values are large.

A common alternative is to focus on failures caused by the combinations of two parameters [24, 10, 16, 9]. But the challenge remains in analyzing the test results to discern the exact cause of failures. Suppose a test run of $10,000$ test cases shows that 100 of the $10,000$ test cases cause the SUT to fail, and each test case consists of 20 parameters. Where should one begin the analysis and debugging process among these 100 failed test cases? Suppose one starts by examining the first failed test case. Which one of the 20 parameters should one examine in detail? Our approach to this

Line	CallerOS	ServerOS	CalleeOS
1	**Mac**	**Lin**	**Mac**
2	Mac	Lin	Win
3	Mac	Sun	Mac
4	**Mac**	**Sun**	**Win**
5	Mac	Win	Mac
6	**Mac**	**Win**	**Win**
7	Win	Lin	Mac
8	**Win**	**Lin**	**Win**
9	**Win**	**Sun**	**Mac**
10	Win	Sun	Win
11	**Win**	**Win**	**Mac**
12	Win	Win	Win

Table 1.1: Test space for a hypothetical VoIP product

challenge is to concentrate on the bad pairs: those pairs of parameter values that always cause the containing test cases to fail.

This approach can be illustrated with a hypothetical test for VoIP software. Assume that bug reports suggest there are three possible error-causing factors, namely, the calling phone, the VoIP server, and the called phone. Suppose these three parameters are CallerOS, ServerOS, and CalleeOS, respectively. Table 1.1 shows the complete test space of 12 rows consisting of the three parameters. Together, the six rows shown in boldface constitute a pairwise test because these six rows include every possible pairwise combination of parameter values. For example, rows 1, 4, 8, and 9 of Table 1.1 cover all possible pairwise combinations of the values of CallerOS and CalleeOS.

If the test results show that the SUT always fails whenever ServerOS equals Lin and CalleeOS equals Mac regardless of the value of CallerOS, we say that this pair of parameter values is a bad pair. It should then be useful information for debugging to

find these bad pairs. Thus, the approach in this thesis concentrates on the analysis of test outputs, unlike pairwise testing, which typically focuses on efficient generation of test inputs. In particular, if there are many test cases resulting in failures, the identification of bad pairs should generate more useful debugging information. To explore these and other related notions, the BadPair framework has been implemented along with three case studies conducted using the framework.

The framework facilitates the debugging process in two ways. The bad pairs analysis helps a debugger to focus on those test cases containing independent bad pairs among all failed test cases. Additionally, by analyzing the characteristics of the two parameter values of an independent bad pair, a debugger can efficiently locate the likely faults in the source code.

The primary contributions of this thesis include the following:

- An automated test framework implementing the proposed approach of bad pairs analysis is presented along with three case studies.
- The study finds that there is a positive relationship between the number of bad pairs and the number of failed test cases.
- The study suggests that there is a positive relationship between the number of bad pairs and the number of faults.
- It appears that filtering of test cases has a significant impact on the bad pairs identified.
- The last case study demonstrates how the BadPair framework can be utilized to facilitate the debugging process in testing an industrial network device.

The remaining Chapters of the thesis are organized as follows. In Chapter 2, key terms are explained. Chapter 3 describes the BadPair framework. Chapters 4, 5, and 6 discuss three case studies using the BadPair framework. The last three chapters contain the related work, conclusion, and suggestions for future work. A published paper related to this thesis will be available at [6].

Chapter 2
Definitions

Several key terms will appear repeatedly throughout this thesis. Therefore, in this section the terms are defined and illustrated with examples.

2.1 Input Table, Results Vector, Test Table

An *input table* is a set of n-tuples, often shown in tabular form. Each row in an input table represents a test case, consisting of n parameters in order, and each column corresponds to one of the n parameters. A *results vector* is a one-dimensional array where each element is either 'P' (pass) or 'F' (fail). A *test table* is a combination of an input table with k rows and a corresponding results vector with k elements. Bad pairs analysis is conducted on test tables. Consider a hypothetical program with three integer inputs: a, b, and c. Figure 2.1(a) shows an input table containing five test cases and a results vector showing the results of executing each test case. Figure 2.1(b) shows the corresponding test table.

2.2 Bad Singleton

The remaining terms are defined with respect to an input table T with n columns and k rows, and a results vector R with k elements. A *singleton* is a value for a specific parameter. In this paper, we represent a singleton visually: as an n-tuple where all the elements except one contain '•'. In the input table in Figure 2.1(a), $(\bullet, 2, \bullet)$ denotes value 2 for parameter b. A *bad singleton* is a singleton which always results in a failure, i.e., whenever test case $T[i]$ contains the singleton then $R[i]$ is 'F'. Figure 2.1(c) shows a bad singleton: in every row in which parameter a has value 1, the result is 'F'. Figure 2.1(a) contains no other bad singletons.

Input Table			**Results Vector**
a	b	c	Pass or Fail
1	1	1	F
1	2	2	F
2	1	2	P
2	2	1	P
2	2	2	F

(a) Input table and results vector

1	1	1	F
1	2	2	F
2	1	2	P
2	2	1	P
2	2	2	F

(b) Test table

a	b	c
1	•	•

(c) A bad singleton

a	b	c
1	2	•
1	1	•
1	•	2
1	•	1
•	1	1
•	2	2

(d) Bad pairs

Figure 2.1: Bad pair analysis examples, part I

a	b	c
1	2	•
1	1	•
1	•	2
1	•	1

(e) Dependent bad pairs

a	b	c
•	2	2
•	1	1

(f) Independent bad pairs

	Failure				Bad Singleton	
Line	Ratio	a	b	c	Index	Type
1	1.0	1	2	•	0	dependent
2	1.0	1	1	•	0	dependent
3	1.0	1	•	2	0	dependent
4	1.0	1	•	1	0	dependent
5	1.0	•	2	2		independent
6	1.0	•	1	1		independent
7	0.5	2	2	•		
8	0.5	2	•	2		
9	0.0	2	1	•		
10	0.0	2	•	1		
11	0.0	•	2	1		
12	0.0	•	1	2		

(g) Failure ratios table for all pairs

Figure 2.2: Bad pair analysis examples, part II

2.3 Pair, Bad Pair, Good Pair

A *pair* is a pair of parameter values. We represent a pair visually: as an n-tuple where all the elements except two contain '•'. For example, in the input table in Figure 2.1(a), $(1, \bullet, 2)$ denotes the pair with value 1 for parameter a and value 2 for parameter c. A *bad pair* is a pair which always results in a failure. For example, Figure 2.1(d) shows all of the bad pairs from Figure 2.1(a). The pair $(1, \bullet, 2)$ is a bad pair because the only row containing it has an 'F' in the results vector. However, $(2, \bullet, 2)$ is *not* a bad pair because it is in two rows, one of which has a 'P'. Finally, a *good pair* is a pair which always results in a pass.

2.4 Dependent Bad Pair, Independent Bad Pair

We divide bad pairs into two types:

1. A *dependent bad pair* is a bad pair which contains one or two bad singletons. Figure 2.2(e) shows the four dependent bad pairs from Figure 2.1(a).

2. An *independent bad pair* is a bad pair which contains no bad singletons. Figure 2.2(f) shows the two independent bad pairs from Figure 2.1(a).

Intuitively, an independent bad pair is a bad pair because of the interaction between the two parameter values, while a dependent bad pair is a bad pair because of the presence of the bad singleton(s) it contains. Our analysis focuses on independent bad pairs.

Figure 2.2(g) summarizes the test results by presenting the failure ratios for each pair in Figure 2.1(a). A bad pair has a failure ratio of 1 while a good pair has a failure ratio of 0.

2.5 Degenerate Cases

There are two degenerate cases:

1. If there is just one parameter ($n = 1$), then each failing row contains a bad singleton. There are no pairs of any kind.

2. If $n = 2$, each row is itself a pair. Every pair is either a bad pair or a good pair; only failure ratios 0 and 1 are possible.

2.6 Sensitivity of Bad Pairs to Change in A Test Table

By definition, the determination of bad pairs is derived from the content of a test table. Therefore, changing the content of a test table will result in change in the independent and dependent bad pairs identified. For example, suppose the following line:

```
2 1 1 P
```

is added to the test table in Figure 2.1(b). Figures 2.3 and 2.4 show the new test table and bad pairs identified consequently. In short, the new added line causes the failure ratio of the pair $(\bullet, 1, 1)$ to drop from 1.0 to 0.5 as shown in line 8 of Figure 2.4(g). In turn, $(\bullet, 1, 1)$ is no longer identified as a bad pair, neither independent nor dependent.

Input Table			**Results Vector**
a	b	c	Pass or Fail
1	1	1	F
1	2	2	F
2	1	2	P
2	2	1	P
2	2	2	F
2	**1**	**1**	**P**

(a) Input table and results vector

1	1	1	F
1	2	2	F
2	1	2	P
2	2	1	P
2	2	2	F
2	**1**	**1**	**P**

(b) Test table

a	b	c
1	•	•

(c) A bad singleton

a	b	c
1	2	•
1	1	•
1	•	2
1	•	1
•	2	2

(d) Bad pairs

Figure 2.3: Sensitivity of bad pairs to change in a test table, part I

a	b	c
1	2	•
1	1	•
1	•	2
1	•	1

(e) Dependent bad pairs

a	b	c
•	2	2

(f) Independent bad pairs

Line	Failure Ratio	a	b	c	BS Index	Type
1	1.0	1	2	•	0	dependent
2	1.0	1	1	•	0	dependent
3	1.0	1	•	2	0	dependent
4	1.0	1	•	1	0	dependent
5	1.0	•	2	2		independent
6	0.5	2	2	•		
7	0.5	2	•	2		
8	**0.5**	•	**1**	**1**		
9	0.0	2	1	•		
10	0.0	2	•	1		
11	0.0	•	2	1		
12	0.0	•	1	2		

(g) Failure ratios table for all pairs

Figure 2.4: Sensitivity of bad pairs to change in a test table, part II

Chapter 3
The BadPair Framework

In this chapter, the design and implementation of the BadPair framework are described. An overview and usage examples of the framework are provided first, followed by a description of the design of the framework. Afterwards, the implementation of the framework is laid out in detail.

3.1 Overview of The Framework

To evaluate our ideas of bad pairs, a harness is needed to conduct experiments, generate test results, and analyze the results to produce viable conclusions. Therefore, we have designed and implemented the BadPair framework. The framework serves several purposes. First, it represents an implementation of the ideas of bad pairs. Secondly, it allows us to understand the general complexity in applying the bad pairs in automated software testing. Further, it demonstrates the feasibility of the bad pairs approach.

The BadPair framework must therefore support several major features to fulfil the desired purposes. For instance, it must be able to generate input test cases in the format as shown in the input table defined in Chapter 2. Also, it has to be capable of generating mutated versions of the targeted source code, which will be referred to as the **gold** code from here on. Finally, it has to be able to filter the test results and analyze either the filtered or original test results and subsequently locate and report the corresponding bad pairs. Note that these three major features are related but nonetheless can be applied independently. For example, the BadPair framework can be directly used to identify and analyze the bad pairs given only a test table, in the absence of test cases and mutated versions of the gold code.

As shown in Figure 3.1, the BadPair framework consists of five components: mutants component, test cases component, auxiliary component, core component, and

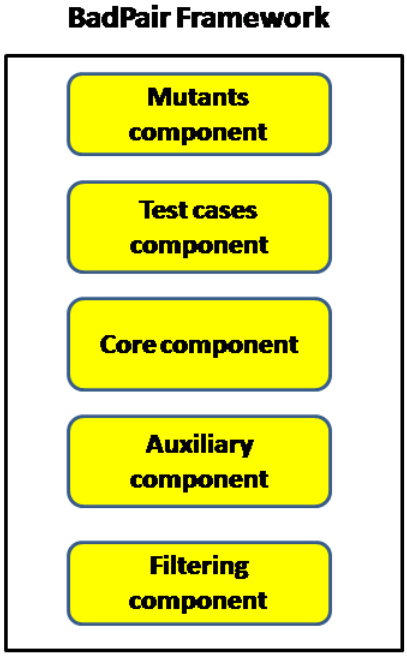

Figure 3.1: Five components of the BadPair framework

filtering component. By breaking down the framework into components, BadPair can be used flexibly in two different ways. First, given only the gold code, one can use the BadPair framework to identify the bad pairs, as shown in Figure 3.2. The result of bad pairs analysis is stored in failure ratios tables, as shown in Figure 2.2(g) of Chapter 2. In addition, visual plots that summarize the relationship between mutants and their corresponding number of bad pairs will be produced. Alternatively, given only a test table without the gold code, one can still use BadPair by directly invoking the core component to identify and analyze the bad pairs, as shown in Figure 3.3. Note that only one corresponding failure ratios table is produced in this case because there is only one test table. If there are many test tables, there will be just as many corresponding failure ratios tables produced.

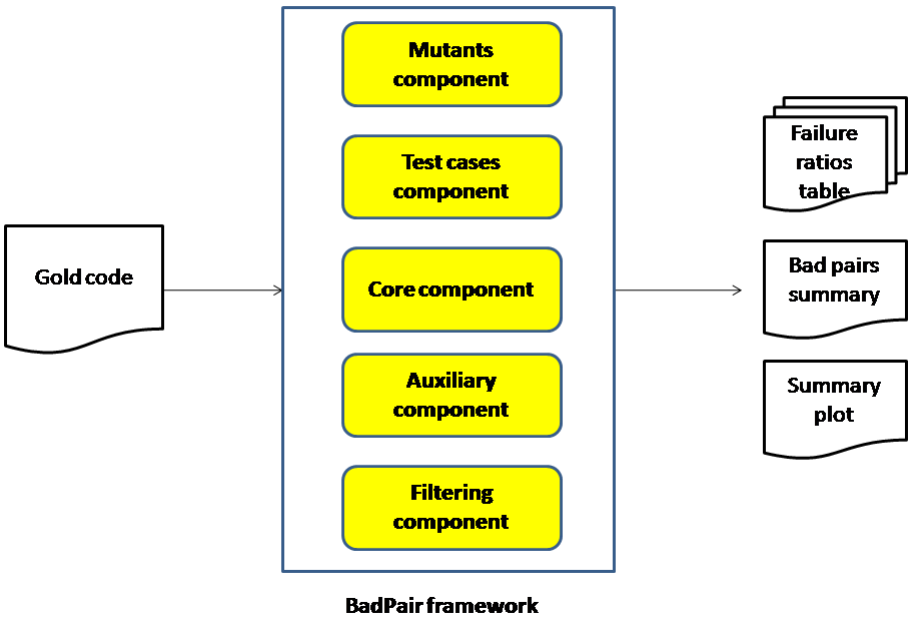

Figure 3.2: Invoking the BadPair framework given only the gold code

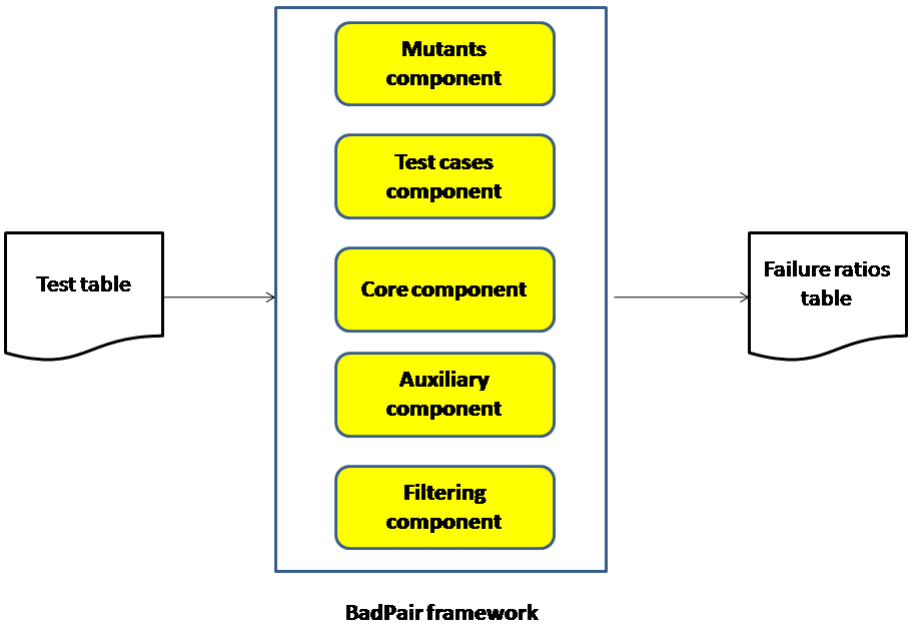

Figure 3.3: Invoking the BadPair framework given only a test table

1	1	1
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2
2	2	1
2	2	2

Figure 3.4: Example of an input table containing 8 test cases

3.2 Examples

3.2.1 No Filtering

Assume there are 8 test cases, as shown in Figure 3.4, to be executed on the gold code and two mutants, M1 and M2. Figures 3.5 and 3.7 show the resulting test tables generated by the BadPair framework from the execution on M1 and M2, respectively.

Based on the two test tables, the BadPair framework then generates two corresponding failure ratios tables, as shown in Figures 3.6 and 3.8 for M1 and M2. The failure ratios table in Figure 3.6 indicates that M1 has 1 independent bad pair, and the failure ratios table in Figure 3.8 indicates that M2 has 2 independent bad pairs. As a summary, BadPair generates a visual plot as shown in Figure 3.9. The plot indicates that 1 mutant has exactly 1 independent bad pair, and 1 mutant has exactly 2 independent bad pairs.

3.2.2 With Filtering

Next, assume the test cases are filtered such that the last test case, '2 2 2', is excluded. In turn, the last line, '2 2 2 P', of the test table for M1 in Figures 3.5 is excluded to produce the filtered test table for M1. Similarly, the last line of the test table for M2 in Figures 3.7 is excluded to produce the filtered test table for M2. Furthermore, based on the filtered test tables, the corresponding failure ratios

1	1	1	P
1	1	2	P
1	2	1	P
1	2	2	P
2	1	1	F
2	1	2	P
2	2	1	F
2	2	2	P

Figure 3.5: The test table from the test run on M1

100.00%	2	*	1
50.00%	2	2	*
50.00%	2	1	*
50.00%	*	2	1
50.00%	*	1	1
0.00%	2	*	2
0.00%	1	2	*
0.00%	1	1	*
0.00%	1	*	2
0.00%	1	*	1
0.00%	*	2	2
0.00%	*	1	2

Figure 3.6: The failure ratios table corresponding to M1

1	1	1	P
1	1	2	F
1	2	1	F
1	2	2	P
2	1	1	F
2	1	2	F
2	2	1	P
2	2	2	P

Figure 3.7: The test table from the test run on M2

100.00%	2	1	*
100.00%	*	1	2
50.00%	2	*	2
50.00%	2	*	1
50.00%	1	2	*
50.00%	1	1	*
50.00%	1	*	2
50.00%	1	*	1
50.00%	*	2	1
50.00%	*	1	1
0.00%	2	2	*
0.00%	*	2	2

Figure 3.8: The failure ratios table corresponding to M2

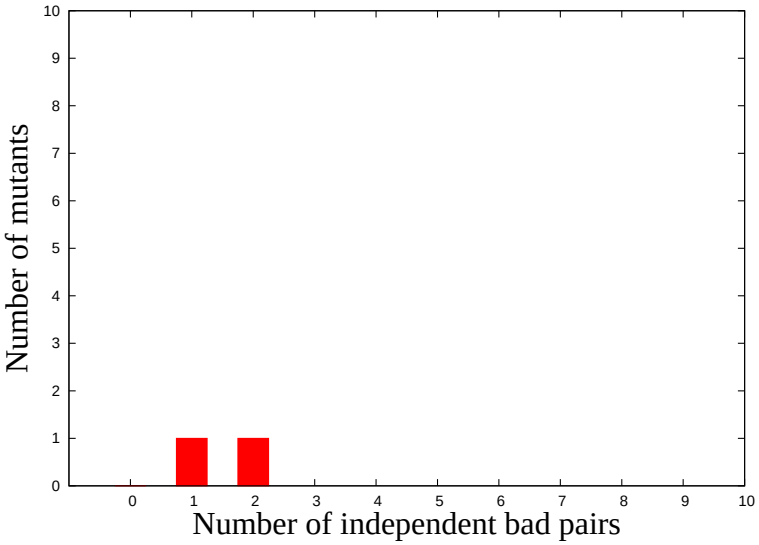

Figure 3.9: The summary plot for the test run on M1 and M2

tables for M1 and M2 are regenerated by the BadPair framework. As a result, the new failure ratios table for M1 indicates that it has 2 independent bad pairs, namely, $(2, 2, \bullet)$ and $(2, \bullet, 1)$. Also, the new failure ratios table for M2 indicates that it has 3 independent bad pairs, namely, $(2, 1, \bullet)$, $(2, \bullet, 2)$, and $(\bullet, 1, 2)$.

Like before the filtering, BadPair also generates a visual summary plot as shown in Figure 3.10. The plot indicates that 1 mutant has exactly 2 independent bad pairs, and 1 mutant has exactly 3 independent bad pairs aftering filtering.

3.3 Design of The Framework

To satisfy the requirements, provide flexibility, and automate the execution flow, the BadPair framework is structured to consist of five components as shown in Figure 3.1. In the following sections, the functionality of each component is described.

3.3.1 Mutants Component

Given the gold code, the mutants component is used to generate its mutated versions. BadPair is designed such that the generation of mutants is separated from the other BadPair components. Thus, any tool can be used to generate mutants of

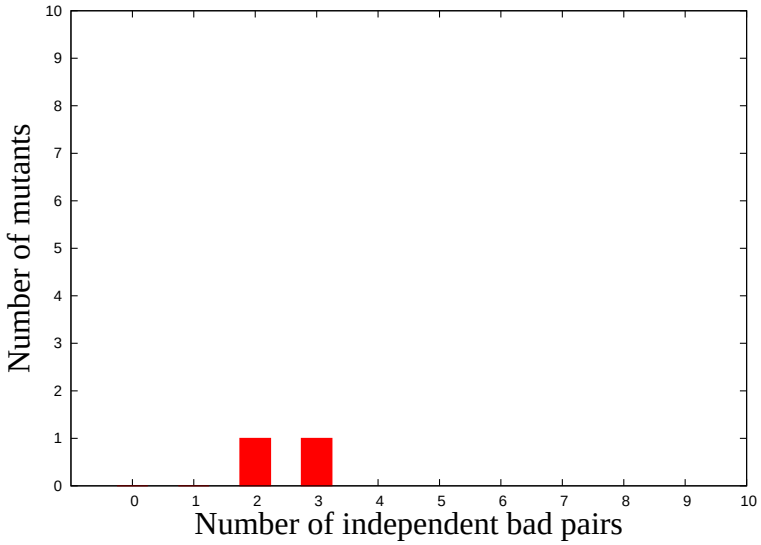

Figure 3.10: The summary plot for the test run on M1 and M2 after filtering

the gold code. The only requirement is that test cases must be able to execute on the gold code and mutants to generate valid test tables for analysis.

3.3.2 Test Cases Component

The primary functionality of the test cases component is to generate test cases to be executed on the gold code and mutants. BadPair permits test cases to be generated by any tool. Regardless of what test cases are generated, the test cases must conform to the format defined for the input tables shown in Figure 2.1(a) of Chapter 2. In addition, the generated test cases must be stored in a log file before they are executed on the gold code and mutants. There are several reasons for this. First of all, the same set of test cases are executed against the gold code and each mutant repeatedly. In addition, test cases may be too large to be held all at once in memory at run time. Lastly, this is a direct result of separating the test case generation from the execution.

3.3.3 Auxiliary Component

The auxiliary component plays a supporting role in BadPair. Its primary functionality is to tie all the components together to automate the execution flow. In addition, it contains any other programs and scripts in the framework that do not

belong to the other components. The following is a list of modules of the auxiliary component:

- Test_one generates one test table from the execution on one mutant
- Test_many generates all corresponding test tables from the execution on all mutants
- config holds the possible values for all input parameters
- convert_to_indexed normalizes test tables

3.3.4 Core Component

The core component provides several key functionalities. First, it controls the execution of test cases on the gold code and mutants. Second, it collects and records the test results. Most importantly, it analyzes the test results to identify the corresponding bad pairs. Additionally, it produces tables of failure ratios based on the test results. Lastly, it generates visual plots that summarize the outcome of the analysis. Consequently, the core component consists of these modules:

- Identification - to identify bad singletons and independent bad pairs
- Frequencies - to generate failure ratios tables
- All frequencies - to create indexed test tables and all failure ratios tables
- Summary - to generate a summary of analysis of bad pairs
- Chart - to create a visual plot of the summary

3.3.5 Filtering Component

The identification and analysis of bad pairs are performed in the context of a given set of test tables. There are times when it is more useful to perform the analysis on subsets of the complete test tables. Thus, the filtering component allows filtering of test tables so that the analysis of bad pairs can be based on some subset of the

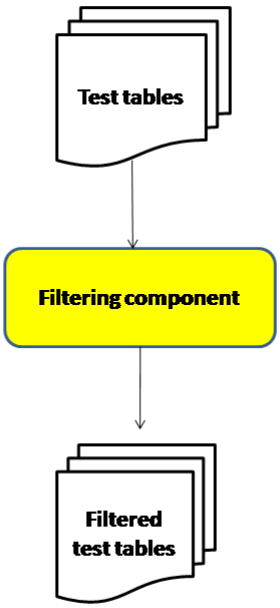

Figure 3.11: Filtering test tables with the filtering component

complete test tables. The design of BadPair allows customized filtering tailored to user-specified filtering requirements. As Figure 3.11 shows, the filtering component takes a set of test tables as input and produces a set of filtered test tables.

3.3.6 Pseudocode of The Execution Flow

As mentioned previously, there are primarily two ways to use BadPair. Figure 3.2 represents the scenario of using BadPair given only the gold code. Figure 3.13 and Figure 3.14 show the corresponding execution flow and execution pseudocode of this scenario, respectively. There are various mutants, each of which has its own source code, a corresponding test table and a failure ratios table; therefore, there is a container folder to hold all mutants and their corresponding generated files. Figure 3.12 shows the structure of this container folder. In addition, for each test execution of the gold code and its mutants, there is a summary and a plot of bad pairs produced. They are stored in file folders also. Figures 3.16 and 3.17 show the structures of these two folders.

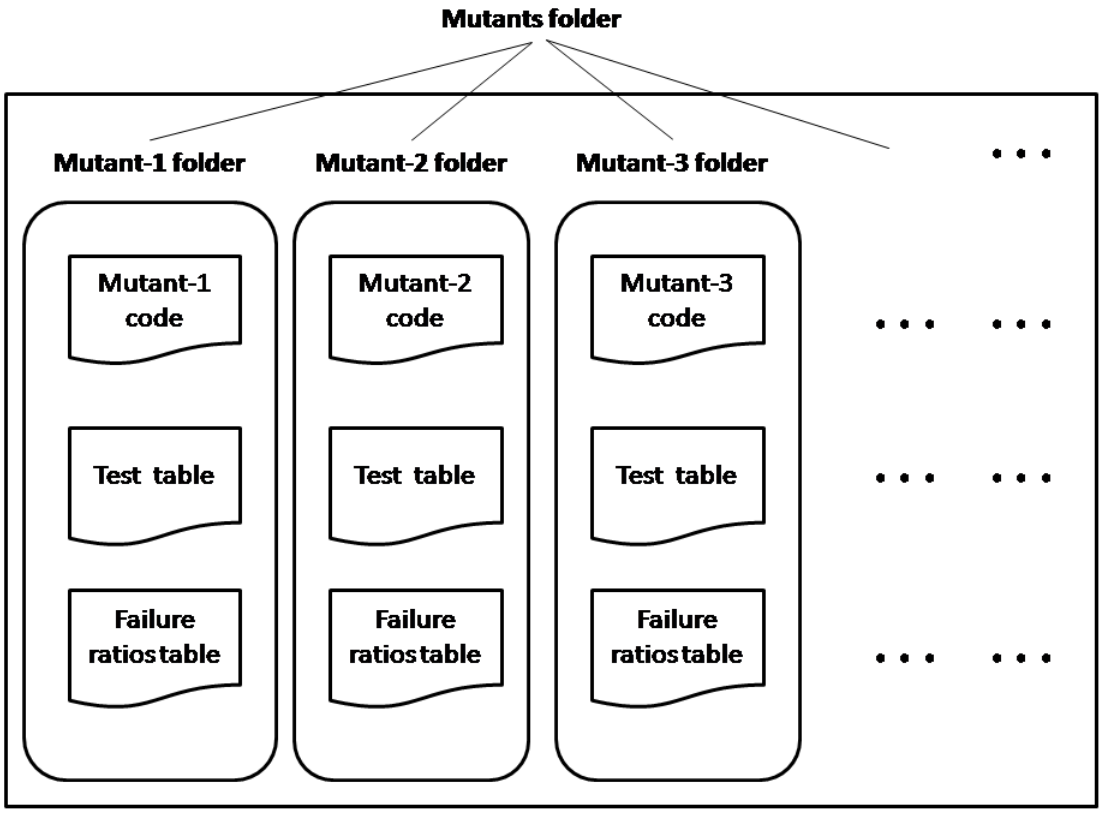

Figure 3.12: File Structure of the mutants folder

Figure 3.3 depicts the alternative scenario of using BadPair given only a test table. Figure 3.15 shows the corresponding execution pseudocode of this scenario.

3.4 Implementation of The Framework

In this section, the implementation of BadPair is presented. BadPair is implemented mainly to be used as a harness to demonstrate our approach, conduct case studies, and validate results. It is implemented in Python with a Ubuntu distribution of Linux as the execution environment. As depicted in Figure 3.1, there are five components in BadPair. Implementation of each of these components are discussed in the next several sections with examples where appropriate.

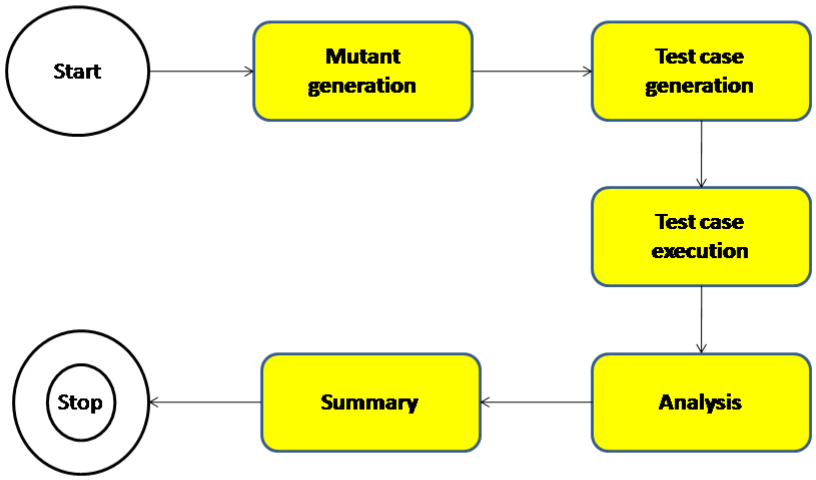

Figure 3.13: Execution flow given only a gold code

3.4.1 Mutants Component

Given a gold code, the mutants component generates mutants of the gold code. The design of BadPair allows any tools to be used to generate mutants. Initially at an earlier stage of developing the BadPair framework, we implemented our own program that generated mutants based on the template and probe methodology as described in a paper by Hoffman et al [14]. Since then, we have switched to a third party tool, muJava [19] [2], because it generates better mutants.

Because muJava can only accept source code in Java as input and can only generate mutants in Java, all gold code in this study has been converted to Java prior to generating mutants. Nevertheless, BadPair framework permits the gold code and mutants in other programming languages besides Java. For example, to execute test on gold code in C, one merely needs to modify the Test_one and Test_many modules of the Auxiliary component. For the purpose of facilitating the presentation and discussion in the rest of this chapter, assume that there are only two mutants, M1 and

```
generate mutants
generate test cases
for each mutant M
        open test table log file L_M
        for each test case t
                run M with input t
                run the gold code with input t
                if M and the gold code produce the same output
                        write t followed by 'P' to L_M
                else
                        write t followed by 'F' to L_M
        close log file L_M
for each log file L_M
        open log file L_M
        open failure ratios table file F
        generate failure ratios from L_M
        write failure ratios to F
        close F
        close L_M
open summary file s
for each F
        open F
                generate bad pairs summary from F
                write the summary to s
        close F
open summary chart file c
generate summary chart data from s
close s
write summary chart data to c
generate chart from c
close c
```

Figure 3.14: Execution pseudocode given only a gold code

open test table log file L

open failure ratios table log file F

generate failure ratios from L

write failure ratios to F

close F

close L

Figure 3.15: Execution pseudocode given only a test table

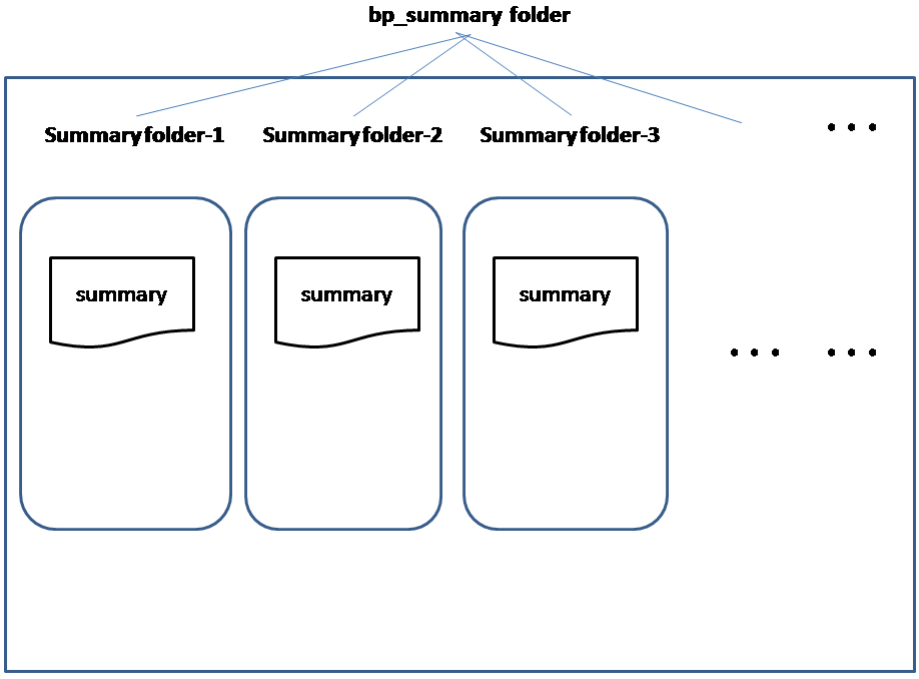

Figure 3.16: File Structure of the summary folder

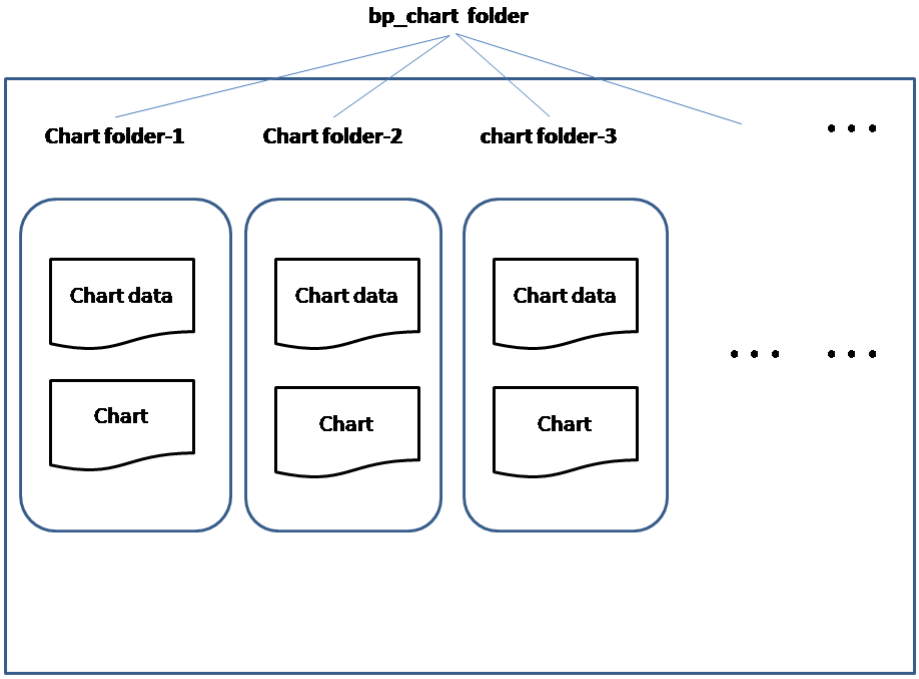

Figure 3.17: File Structure of the chart folder

M2, as mentioned earlier created and ready to be executed against test cases.

3.4.2 Test Cases Component

The sole purpose of the test cases component is to generate the desired test cases to be executed. As mentioned in the previous section, the design of BadPair framework allows test cases to be generated by any tool as long as they conform to the format defined for input tables in Figure 2.1(a) of Chapter 2. The test cases generated are to be stored in a file, say, `input.txt`.

Assume that there are 8 test cases generated and stored in the input table as shown in Figure 3.4. Each line of the input table represents one test case consisting of 3 ordered input parameters, and each column represents the value of an input parameter. For instance, the first line is a test case that has the values 1, 1, and 1 for the first, second, and third parameters, respectively, while the last line is a test case that has the values 2, 2, and 2 for the first, second, and third parameters, respectively. There are in total 8 test cases because each one of the 3 input parameters can take

on the value of either 1 or 2.

3.4.3 Auxiliary Component

The auxiliary component contains any other modules and scripts in the framework that do not belong to the other four components. In addition, It contains scripts used to tie all components together to automate the execution flow. There are several modules contained in this component:

- Test_one.java
- Test_many.py
- config.py
- convert_to_indexed.py

3.4.3.1 Test_one.java

Lines of code: 49

Number of functions: 1

Given the gold code, a mutant, and a file containing the test cases, this module is responsible for executing all given test cases on the gold code and the mutant. It is also responsible for recording test results from executing the test cases. In turn, it produces a test table corresponding to the test run on the given mutant and gold code. Note that this module is written in Java because for the purpose of this study both the given gold code and mutants are in Java.

For example, Figure 3.5 shows the resulting test tables from the execution of the 8 test cases of Figure 3.4 on the gold code and M1, and Figure 3.7 shows the resulting test tables from the execution on the gold code and M2. Note that if the test cases were executed on the gold code against the gold code itself, the resulting test table would have `P` in every line.

```
token_list =
        [
                ['1','2'],
                ['1','2'],
                ['1','2']
        ]
```

Figure 3.18: Example of config.py

3.4.3.2 Test_many.py

Lines of code: 14

Number of functions: 0

This module is responsible for generating test tables corresponding to the given mutants. Given a list of `paths` to mutants, it repeatedly invokes `Test_one.java` for each mutant in the list. It also records each test table in a log file located in the same given `path` as the corresponding mutant.

3.4.3.3 config.py

This module contains a list, `token-list`, that describes all possible values of all input parameters. The list is a nested list where each sub-list corresponds to one input parameter, and each sub-list depicts all possible values of the corresponding input parameter. Figure 3.18 shows the token-list that corresponds to the input table of 8 test cases shown in 3.4.

3.4.3.4 convert_to_indexed.py

Lines of code: 25

Number of functions: 0

Given a list of files, each of which contains a test table, such as those produced by `Test_many.py`, this module normalizes the test tables by transforming them into indexed formats required before the analysis of bad pairs can be performed. Indices of input parameters are generated based on the token-list contained in the `config.py` module. That is, the first possible value of an input parameter has the index 0, the

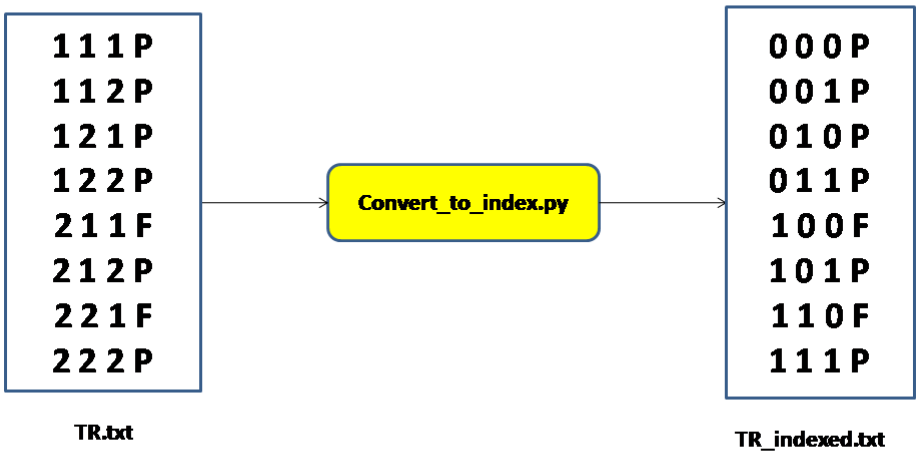

Figure 3.19: Example of converting a test table to an indexed test table

second possible value of an input parameter has the index 1, and so forth. Each converted test table is recorded in a new file located in the same directory as the given file. For instance, based on the token-list given in Figure 3.18, the following line in the test table:

```
2 2 1 F
```

is therefore converted to:

```
1 1 0 F
```

Figure 3.19 shows an example of such conversion where the file, `TR.txt`, containing the test table in Figure 3.5 is converted to an indexed test table, which is written to another file, `TR_indexed.txt`.

3.4.4 Core Component

There are five modules in the core component and each module is further broken down into various functions. These five modules, together with its main functionality, implemented in Python are as follows:

- bad_pairs.py identifies bad singletons and bad pairs
- gen_frequencies.py generates a failure ratios table
- find_bp.py creates indexed test tables and all failure ratios tables
- sum_bp.py to generates a summary of analysis of bad pairs
- build_chart.py to creates a visual plot of the summary

3.4.4.1 bad_pairs.py

Lines of code: 252

Number of functions: 10

The main purpose of the module is to identify bad singletons and bad pairs. Among all functions implemented in the module, the most important ones are as follows:

- count_singletons:

  This function calculates, for each singleton, the number of passed test cases, the number of failed test cases, and the pass ratio among all test cases that contain the singleton. The calculated information is recorded in a list, which is returned as a result.

- generate_bad_singletons:

  This function takes a list of singletons as input, and identifies all bad singletons contained in the list of singletons. It returns a list of these identified bad singletons.

- count_pairs:

  This function calculates, for each pair, the number of passed test cases, the number of failed test cases, and the pass ratio among all test cases that contain the pair. The calculated information is recorded in a list, which is returned as a result.

- generate_bad_pairs:

  This function takes a list of pairs and failure threshold as input, and identifies all bad pairs. It returns a list of these identified bad pairs. The threshold is 1.0 based on the definition that a bad pair is a pair where 100 percent of the time the pair always results in a failed test case. Thus, if the definition of a bad pair is changed to a pair where at least 90 percent of the time the pair results in a failure, the failure threshold should be 0.9 instead.

3.4.4.2 gen_frequencies.py

Lines of code: 247

Number of functions: 6

This module is in charge of generating a sorted failure ratios table and writing it to a specified log file. For example, Figure 3.6 shows the sorted failure ratios table that corresponds to the test table in Figure 3.5.

Among all functions implemented in the module, the most important ones are as follows:

- gen_bp_frequencies:

  This functions uses the bad_pairs.py module to generate an unsorted failure ratios table. The generated table is held in memory.

- build_pair_frequency:

  This function takes an unsorted failure ratios table, sorts the table on failure ratios, and writes the sorted table out to a specified log file in the pre-defined format for the failure ratios table.

- locate_bad_pairs:

  This function identifies bad pairs based on the execution on one mutant. It does so by calling various functions in the bad_pairs.py module before invoking build_pair_frequency function.

3.4.4.3 find_bp.py

Lines of code: 52

Number of functions: 2

This module is responsible for creating indexed test tables and all failure ratios tables. It contains the following two functions.

- build_indexed_results:

  This function builds two lists. The first list is a a list of test cases. The second list is a nested list where each sub-list is a results vector. Note that there is only one list of test cases because all mutants are executed against the same set of test cases, while the second list is nested because each sub-list corresponds to a results vector from execution on one mutant.

- find_bp:

  This function writes each test table to its corresponding log file by first invoking the build_indexed_results function and subsequently using the gen_frequencies.py module.

3.4.4.4 sum_bp.py

Lines of code: 53

Number of functions: 1

Given a set of files containing failure ratios tables, this module generates a summary table of analysis of bad pairs, and writes the summary table to a specified file. Figure 3.20 shows the summary table that corresponds to the test tables of M1 and M2. Each line in the summary file represents the analysis result of one mutant and each line consists of four entries. For instance, the first line indicates there are 2 independent bad pairs, 0 bad singletons, and 12 pairs from the test run on M2. The second line indicates there are 1 independent bad pair, 0 bad singletons, and 12 pairs from the test run on M1. There is one function, sum_bp, in this module.

- sum_bp:

2	0	12	M2
1	0	12	M1

Figure 3.20: The summary table for the test run on M1 and M2

Given a file containing a failure ratios table, this function calculates the total number of independent bad pairs and the total number of bad singletons.

3.4.4.5 build_chart.py

Lines of code: 55

Number of functions: 1

This module is responsible for generating the raw data that can be used to create a visual plot of the summary based on the result of bad pairs analysis. The raw data generated is written to a file as a result. Figure 3.21 shows the chart table that corresponds to the bad pairs analysis for the test run on M1 and M2. Each line in the chart table consists of two integers. The first integer indicates the number of independent bad pairs, and the second integer indicates the number of those mutants that have the corresponding number of independent bad pairs. For example, the first line in Figure 3.21 indicates that no mutant has no independent bad pairs. The second line indicates that 1 mutant has exactly 1 independent bad pair. The third line indicates that 1 mutant has exactly 2 independent bad pairs. Figure 3.9 shows the corresponding summary plot generated.

There is one function, build_chart_data, in this module.

- build_chart_data:

  Given a file containing a summary table of analysis of bad pairs, this function generates the underlying raw data that can be used to generate a visual plot of the summary and writes the raw data to a file.

0 0

1 1

2 1

Figure 3.21: The chart table generated for the test run on M1 and M2

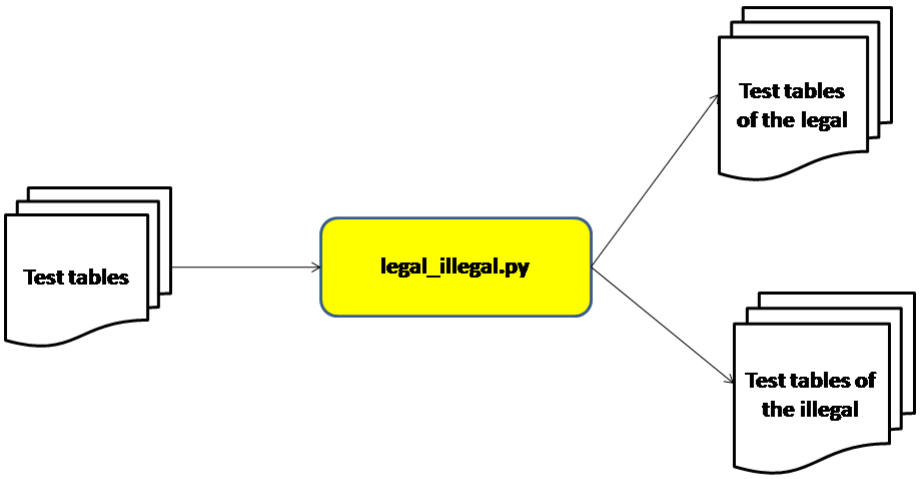

Figure 3.22: Filtering test tables with legal_illegal_.py

3.4.5 Filtering Component

The filtering component filters test tables to produce partial test tables. For instance, the python module, legal_illegal.py, can be applied to a set of test tables to break them into two subsets of test tables, one subset for the legal test cases and the other for the illegal test cases, as shown in Figure 3.22. If legal_illegal.py is applied to the test table of Figure 3.5, it will produce the two subsets of partial test tables shown in Figure 3.23 and Figure 3.24.

1	1	1	P
1	2	2	P
2	1	2	P
2	2	1	F
2	2	2	P

Figure 3.23: A partial test table after filtering the test table of M1

1	1	2	P
1	2	1	P
2	1	1	F

Figure 3.24: Another partial test table after filtering the test table of M1

3.5 BadPair Framework and The Case Studies

In the next three chapters, three case studies will be presented. The first two case studies incorporate generation of test cases, mutation testing, and filtering of test cases to determine the effects on the number of bad pairs resulted from changing the contents of input tables. The last case study demonstrates how one can utilize the BadPair framework to analyze bad pairs given only a test table in the absence of the gold code and mutants. Also, alternative definitions of the bad pairs are explored where a threshold is used to define "nearly bad pairs."

In doing the three case studies, we aim to address several immediate aspects of bad pairs:

1. Number of independent bad pairs
2. The effect of filtering test cases on independent bad pairs
3. Frequency of independent bad pairs
4. The effect of seeded faults on independent bad pairs
5. The effect of failure thresholds on the number of independent bad pairs

6. The effect of number of failed test cases on the number of independent bad pairs

Ultimately, we expect to benefit from the three case studies in achieving these longer term goals of bad pairs:

1. The assessment and improvement of the BadPair framework
2. The value of identifying independent bad pairs in software testing
3. The relationship between independent bad pairs and seeded faults
4. The relationship between independent bad pairs and failed test cases

Chapter 4
Case Study: Bad Pairs in The Triangle Program

The first case study involves testing a well known short program, *Triangle* [17], as the gold code. This case study serves several purposes. First, it is used to validate and improve the BadPair framework. Second, it demonstrates an application of BadPair. Lastly, it answers and/or provides insights to our research questions.

4.1 The Triangle Gold Code

Given three integers as the values to the three input parameters, Triangle determines whether the three integers constitute a valid triangle. Each one of the three integers represents the length of one of the three sides of a triangle. If the three given integers do not constitute a valid triangle, the string `"illegal"` is returned as a result indicating an invalid triangle. Otherwise, Triangle returns one of the three strings, `"equilateral"`, `"isosceles"`, or `"scalene"`, indicating the precise triangle type. Figure 4.1 shows the complete Triangle source code in Java.

4.2 Test Setup

The detailed execution flow of this case study is depicted in the pseudocode shown in Figure 3.14 in Chapter 3. In short, there are five major steps involved in the execution flow as depicted in Figure 4.2: Mutant generation, test case generation, test case execution, bad pairs analysis, summary. Given the non-mutated Triangle code, used as the gold code for the test oracle, 213 mutants are generated by muJava. Note that the original source code of Triangle is written in C. We have manually translated it to be in Java because muJava can only operate on source code written in Java. Each one of these mutants contains exactly one single seeded fault. For example, the `&&` conditional operator in line 36 of Figure 4.1 is replaced with the `||` conditional operator by muJava to create one of the mutant. Each application

```
1 public static String triangle(
2   int side1, int side2, int side3)
3 {
4   int triang;
5   if (side1 <= 0 || side2 <= 0 || side3 <= 0) {
6       return "illegal";
7   }
8   triang = 0;
9
10  if (side1 == side2) {
11      triang = triang + 1;
12  }
13  if (side1 == side3) {
14      triang = triang + 2;
15  }
16  if (side2 == side3) {
17      triang = triang + 3;
18  }
19
20  if (triang == 0) {
21      if (side1 + side2 <= side3 ||
22          side2 + side3 <= side1 ||
23          side1 + side3 <= side2) {
24          return "illegal";
25      } else {
26          return "scalene";
27      }
28  }
29
30  if (triang > 3) {
31      return "equilateral";
32  } else if (triang == 1 && side1 + side2 > side3) {
33      return "isosceles";
34  } else if (triang == 2 && side1 + side3 > side2) {
35      return "isosceles";
36  } else if (triang == 3 && side2 + side3 > side1) {
37      return "isosceles";
38  }
39
40  return "illegal";
41 }
```

Figure 4.1: The Triangle “gold” source code

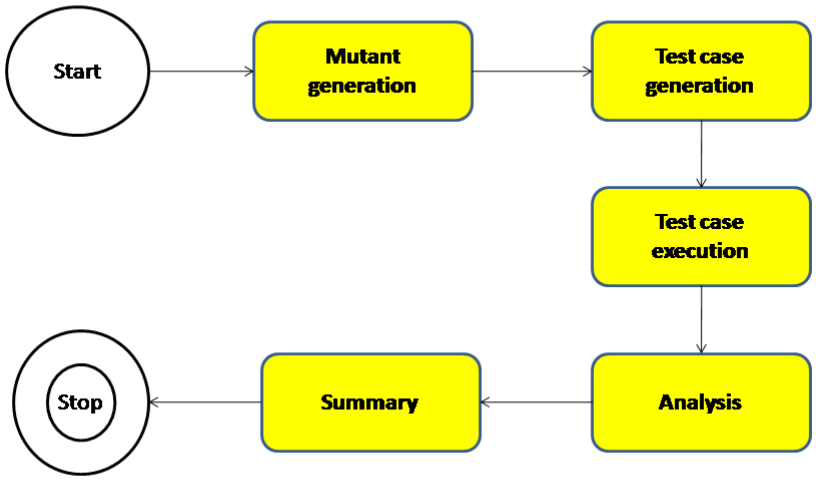

Figure 4.2: Execution flow of the triangle case study

of muJava on the Triangle gold code creates 213 mutants because muJava creates one mutant by replacing one operator in the gold code each time. An operator can be a conditional operator, relational operator, arithmetic operator, or any other type of operators supported by muJava. Therefore, given the same gold code, each application of muJava creates a fixed number of mutants. In addition, there will be 213 test tables in 213 log files, one for each mutant, produced as a result of a complete test run on all mutants.

For evaluating the BadPair framework, YouGen [22, 12] has been chosen as the vehicle for the generation of test cases. Given a user-specified control file, YouGen generates the tailored test cases as depicted in Figure 4.3. Each test case in the input table is a 3-tuple and consists of three ordered input parameters. Each input parameter can be one of the values 0, 1, 2, 3, 4, or 5. Therefore, $6 \times 6 \times 6 = 216$ test cases are generated. Further, each one of the 213 log files produced will contain 216 3-tuples followed by `'P'` or `'F'` as a result of a complete test run on all mutants.

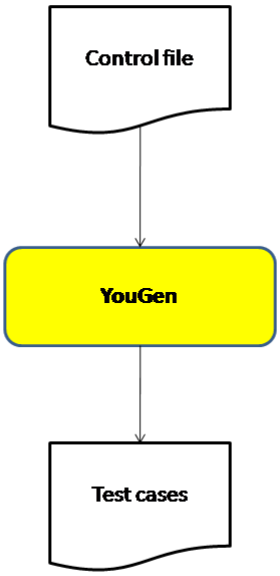

Figure 4.3: Given a control file, YouGen generates a corresponding set of test cases

4.3 Results

In this section, two sets of results are presented respectively, followed by discussion about the results. The first set covers all mutants with one seeded fault, while the second set covers all mutants with two seeded faults.

4.3.1 Single Mutation: Single Seeded Fault Per Mutant

The initial results from executing the complete set of test cases showed that there were no bad pairs at all among the test results from all mutants. This finding consequently led to the development of the filtering component in the BadPair framework. As explained in Chapter 3 the identification and analysis of bad pairs are performed in the context of a given set of test tables. There are times when it is more useful to perform the analysis on subsets of the complete test tables. In this case, no bad pairs were found because the complete set of test cases included test cases representing illegal triangles, which masked the effect of test cases representing legal

triangles. Masking happens when, for instance, the == conditional operator in line 10 of Figure 4.1 is replaced by the > conditional operator. That is,

```
if (side1 == side2) {
```

is replaced by

```
if (side1 > side2) {
```

The net effect of this seeded fault is that a scalene triangle with $a > b, e.g.(4, 2, 3)$, is classified as isosceles. The mutant's test table would contain an entry:

4 2 3 F

Hence, $(4, 2, \bullet)$ would have been identified as a bad pair. Nonetheless, it turned out that not all test cases containing $(4, 2, \bullet)$ failed. For example, $(4, 2, 0)$ passed, because it was correctly classified as illegal by the mutant; hence, the line of code containing the seeded fault was masked and never executed. The same masking pattern recurred for many mutants. Many pairs would have been identified as bad pairs if only test cases of legal triangles were considered. In light of this finding, test tables were filtered so that only those representing legal triangles were included in the subsequent analysis and summary in this chapter. Figure 4.7 depicts the execution flow with the filtering step incorporated.

Figure 4.5 shows the summary plot of 213 test tables from executing test cases of legal triangles on all 213 mutants with a single seeded fault. The X axis is the number of independent bad pairs and the Y axis is the number of mutants with the corresponding number of independent bad pairs. There are, for example, 44 mutants that have 8 independent bad pairs, and 103, almost a half, of the 213 mutants have one or more than one independent bad pair.

In addition, we repeated the test using pairwise testing where only those test cases representing a two-cover of the three input parameters were executed. Figure 4.6 summarizes the test results. In comparison with Figure 4.5, independent bad pairs are distributed more evenly across the 213 mutants while the number of independent bad pairs stays roughly the same.

According to the definition of bad pairs presented in Chapter 2, a bad pair has a failure ratio equal to 1.0. There may be times that identifying nearly bad pairs using

a failure ratio threshold less than 1.0 is of interest. For example, when there are no bad pairs at all even after filtering test cases one may be interested in identifying those pairs that fail 90 percent of the time instead. Therefore, we have repeated the same test run with various thresholds.

At threshold equal to 0.9, there are still no bad pairs at all when run against the complete set of test cases. At thresholds equal to 0.8 and 0.7, there are 49 mutants that have one or more independent bad pair when run against the complete set of test cases. When run against only the test cases of legal triangles, the numbers of independent bad pairs do not change until the threshold is dropped to 0.7.

Another interesting aspect is the relationship between the number of failed test cases and the number of independent bad pairs. Intuitively, one would expect there is a positive relationship between the two. To derive the relationship, the number of failed test cases of each one of the 213 mutants is correlated with the corresponding number of independent bad pairs of each one of the 213 mutants. Figure 4.4 shows that the number of failed test cases and the number of independent bad pairs are positively correlated with R-square correlation coefficient equal to 0.75. Note that two outliers have been removed from the 213 points to obtain the correlation coefficient.

4.3.2 Double Mutation: Two Seeded Faults Per Mutant

A mutant with double seeded faults is one that, for instance, has line 10 of Figure 4.1:

```
if (side1 == side2) {
```

replaced by:

```
if (side1 > side2) {
```

and line 13:

```
if (side1 == side3) {
```

replaced by:

```
if (side1 < side3) {
```

These mutants with double seeded faults (double-mutation mutants) can be created by applying muJava twice. At first muJava is applied to the Triangle gold code,

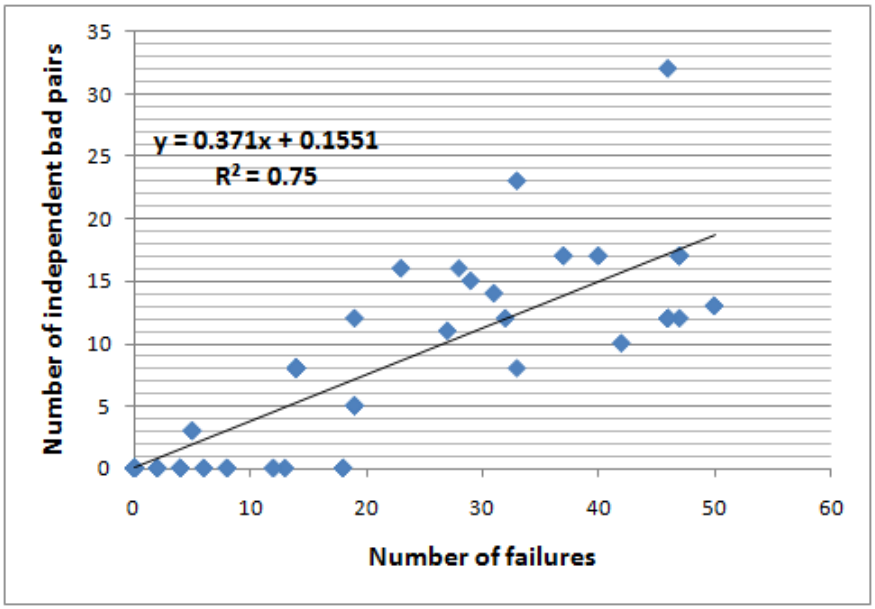

Figure 4.4: A strong positive relationship exists between the number of failures and the number of independent bad pairs with R-square correlation coefficient equal to 0.75 based on 211 data points.

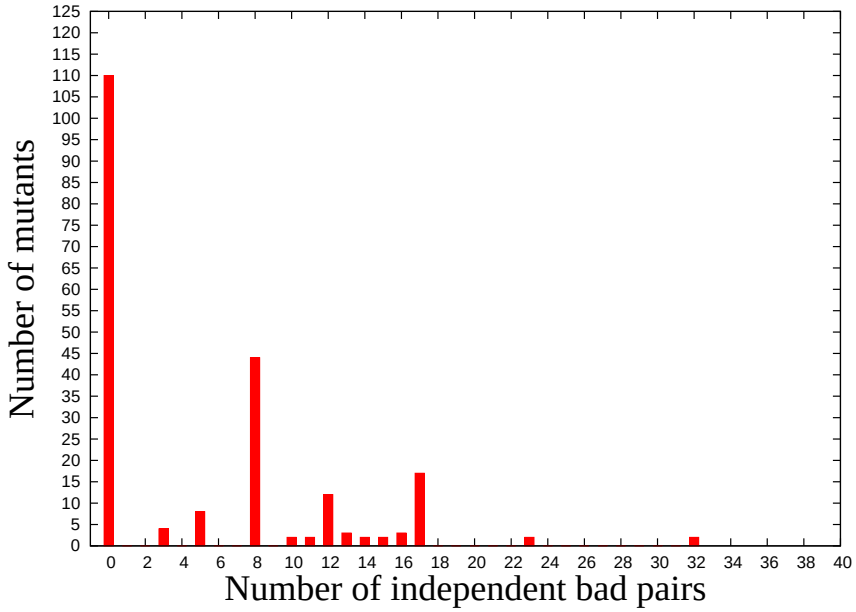

Figure 4.5: Summary plot of single mutation, legal triangles

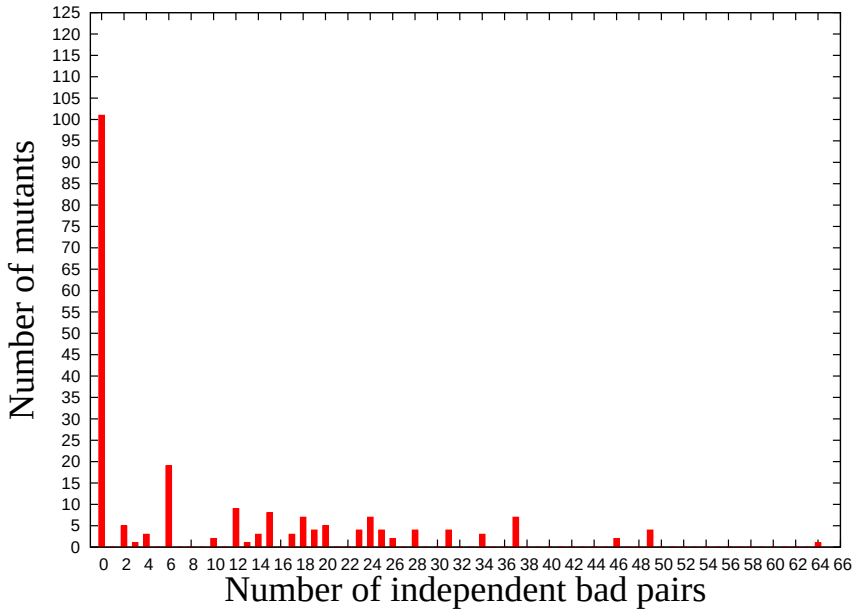

Figure 4.6: Summary plot of single mutation, 2-cover input parameters, legal triangles

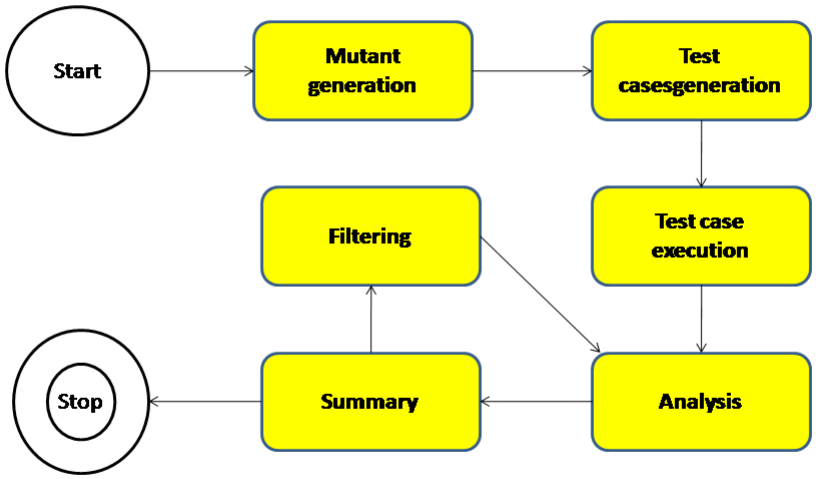

Figure 4.7: Execution flow with filtering of the triangle case study

and then muJava is applied again on the source code of a selected single-mutation mutant. Note that the gold code used as the test oracle for double-mutation mutants is the same non-mutated Triangle gold code. The purpose of testing on double-mutation mutants is to incorporate fault-interactions [15] in this case study.

Three sets of double-mutation mutants have been created for the case study. The three sets are created by applying muJava to three selected single-mutation mutants, each of which has a different mutation operator used by muJava in creating mutants. The three selected operators are conditional operator replacement (COR, such as `&&`, `||`, relational operator replacement (ROR, such as `>`, `<`, `==`, and arithmetic operator replacement (AOR, such as `+`, `-`). Each set consists of 212 mutants rather than 213 because, in one of the 213 mutants, the second mutation negates the first mutation making that mutant the same as the gold code.

Figure 4.8 shows the summary plot from the COR set of 212 double-mutation mutants. The test results indicate that only 22 out of the 212 mutants do not have any independent bad pairs, while 103 of them have 8 independent bad pairs. As for the ROR set, Figure 4.9 shows the summary plot from the ROR set. It suggests that 205 out of the 212 mutants have at least one independent bad pair, and 76 of them have 11 independent bad pairs. Lastly, the summary plot for the AOR set is shown in Figure 4.10. Almost half of the mutants, 94 out of 212, have 8 independent bad pairs, and 200 of them have at least one independent bad pairs.

4.4 Discussion

Several observations are worth noting from the case study. When test results from execution against all test cases are considered and analyzed, there are no bad pairs, independent or dependent, at all. However, when test results are filtered such that only test cases of legal triangles are considered, almost half, 103 out of 213, of the single-mutation mutants have at least one independent bad pair. This finding suggests that filtering of test cases can have a material effect on bad pairs analysis. Second, by lowering the failure thresholds for identifying bad pairs, one can locate nearly bad pairs, which can be useful alternatives when no bad pairs exist at the

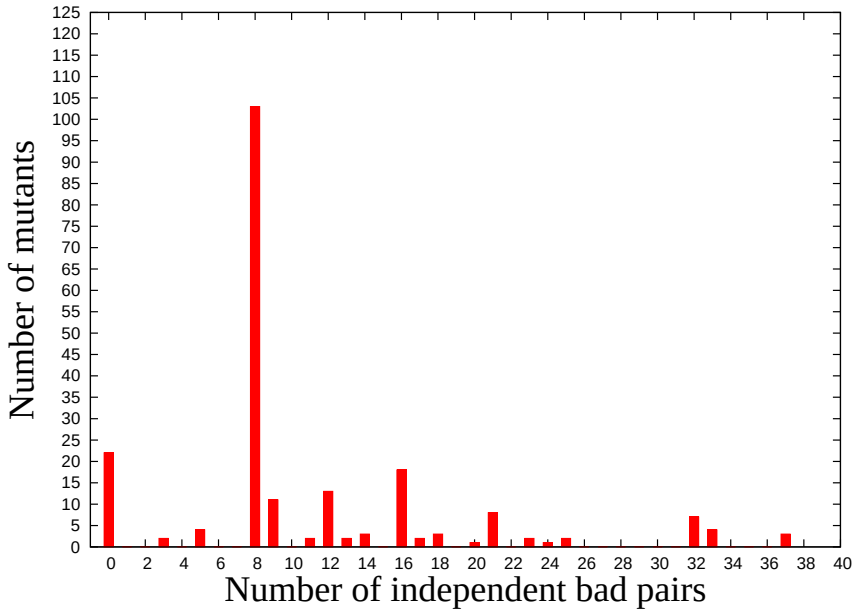

Figure 4.8: Double Mutation COR

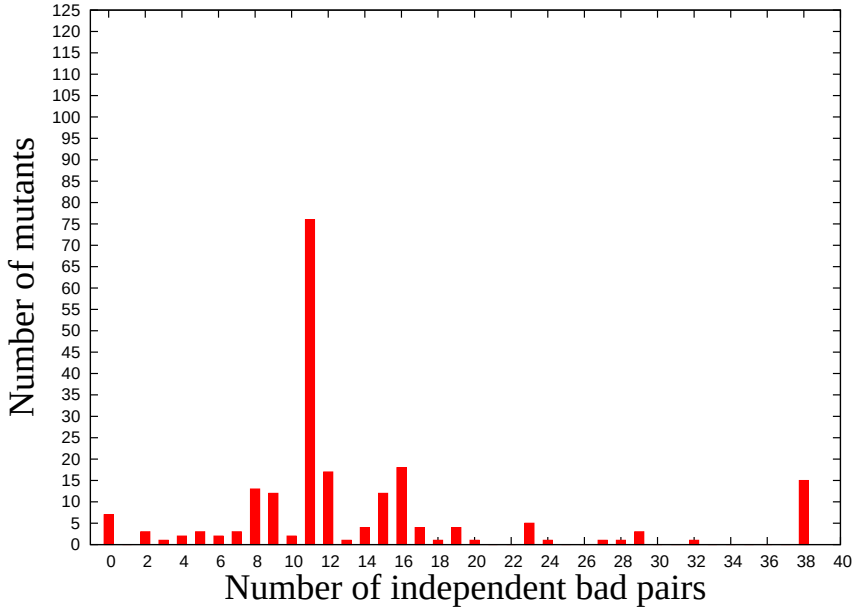

Figure 4.9: Double Mutation ROR

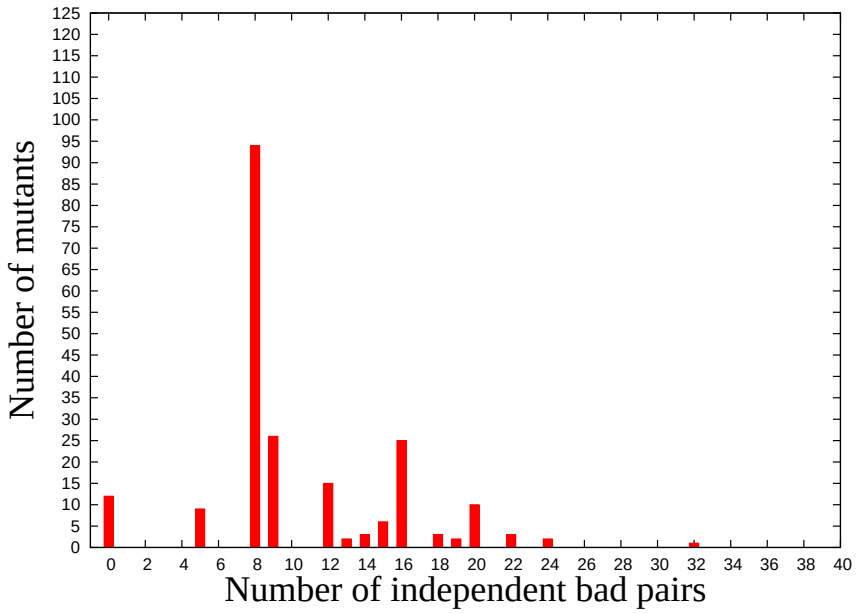

Figure 4.10: Double Mutation AORB

default failure ratio of 1. Third, in comparing single mutation and double mutation, the number of mutants with no independent bad pairs decreases significantly when mutants contain two seeded faults. Furthermore, the overall number of independent bad pairs increases drastically when mutants contain two seeded faults. This suggests that a potential relationship exists between the number of seeded faults and the number of independent bad pairs. Lastly, the data from single-mutation mutants suggests that there is a positive relationship between the number of failed test cases and the number of independent bad pairs with R-square correlation coefficient equal to 0.75.

Chapter 5
Case Study: Bad Pairs in TCAS

For the second case study, the source code of a Traffic Collision Avoidance System (TCAS) obtained from the Siemens Corporate Research [1] is used as the gold code. The purposes of the second study are in principle the same as those of the first case study. An additional intent is to apply the BadPair framework to a larger and more complex program than Triangle.

5.1 The TCAS Gold Code

Like the Triangle case study, the source code of TCAS has been manually converted from C to Java so that muJava can be used to create mutants. 250 mutants have been generated by muJava and used in this case study. Unlike Triangle, the gold code of TCAS is larger and more complex. It has more lines of code; it has 12 input parameters in comparison with 3 in Triangle; each of the 12 parameters can take on many possible values. Therefore, there are significantly more parameter pairs in TCAS. For instance, if each of the 12 parameters are restricted to 10 values, there will be 6600 $(12 * 11/2 * 10 * 10)$ parameter pairs, while Triangle will only have 300 $(3*2/2*10*10)$ parameter pairs. Similarly, TCAS can easily have significantly more test cases than Triangle.

5.2 Test Setup

The execution flow of this case study is as depicted in Figure 5.1. It is basically the same as the Triangle case study, except that there is no filtering of any sort involved. In addition, it is impractical to execute the full cross product of test cases $(1,000,000,000,000$ of test cases if there are 10 possible values for each input parameter) for the purpose of this case study. To reduce the number of test cases, we have selected 3 to 5 values for each input parameter. However, this still would

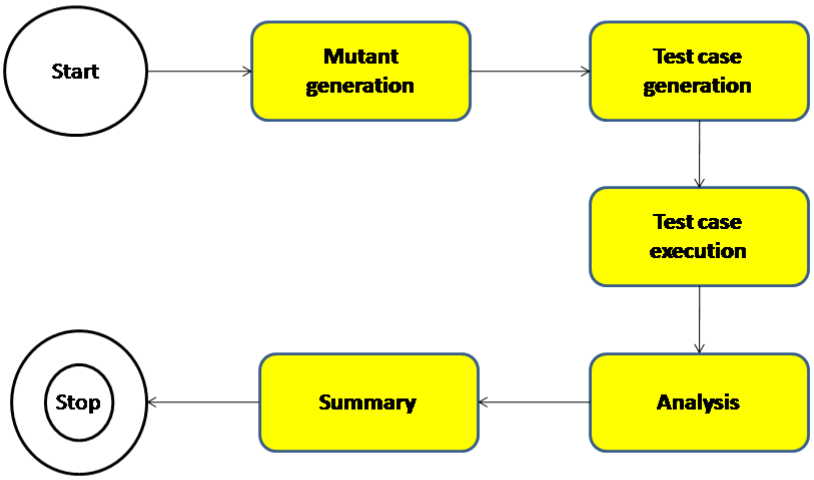

Figure 5.1: Execution flow of the TCAS case study

generate 6,220,800 test cases, again impractical to execute for this study. Thus, we have applied pairwise generation of test cases using YouGen to bring the number of test cases down to 34.

5.3 Results

In this section, two sets of results are presented respectively, followed by discussion about the analysis of bad pairs. The first set covers those mutants with one seeded fault, while the second set covers those mutants with two seeded faults.

5.3.1 Single Mutation: Single Seeded Fault Per Mutant

Figure 5.2 shows the summary plot of the test run from executing against the 34 pairwise test cases on all 250 mutants, each one of which contains a single seeded fault. The X axis is the number of independent bad pairs and the Y axis is the number of mutants with the corresponding number of independent bad pairs. There

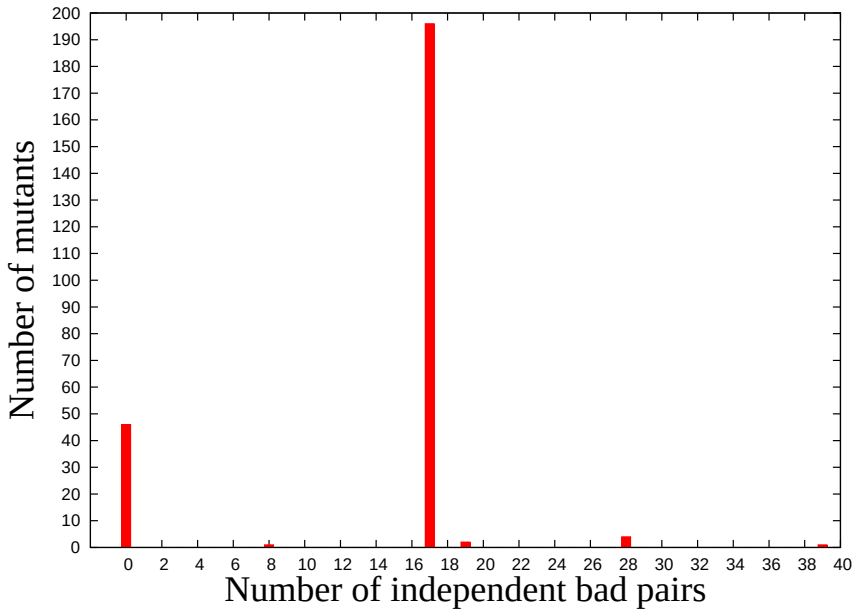

Figure 5.2: TCAS: single mutation with pairwise inputs. There are 204 out of the 250 mutants that have at least 8 independent bad pairs, and 196 out of the 250 mutants have 17 independent bad pairs.

are 204 out of the 250 mutants that have at least 8 independent bad pairs, and 196 out of the 250 mutants have 17 independent bad pairs. Only 46 mutants do not have any independent bad pair.

As in the case study of Triangle, we have repeated the experiments using various thresholds of failure ratio. It turns out that no changes in the independent bad pairs are observed when the threshold is lowered to 0.9, 0.8, 0.7, or 0.6. At threshold 0.5, however, the number of independent bad pairs more than doubles: 196 out of the 250 mutants have 41 independent bad pairs as shown in Figure 5.3.

Also, the number of failed test cases of each one of the 250 mutants is correlated with the corresponding number of independent bad pairs of each one of the 250 mutants. Figure 5.4 shows that the number of failed test cases and the number of independent bad pairs are positively correlated with a high R-square correlation coefficient equal to 0.9466. Note that six outliers have been removed from the 250 points to obtain the correlation coefficient.

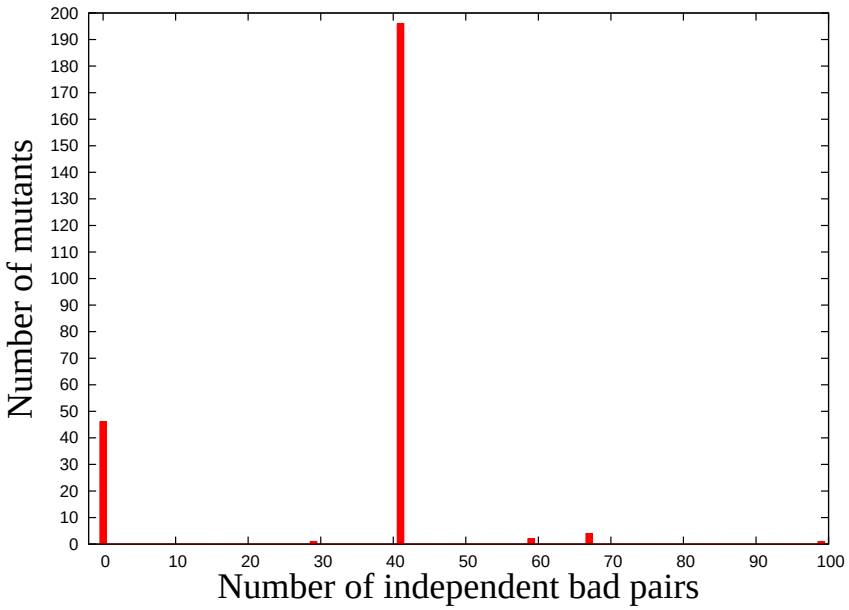

Figure 5.3: TCAS: single mutation with failure threshold at 0.5. The number of independent bad pairs increases more than two folds: 196 out of the 250 mutants have 41 independent bad pairs.

5.3.2 Double Mutation: Two Seeded Faults Per Mutant

Three sets of double-mutation mutants have been created for the case study. As in the case study of Triangle, these three sets of double-mutation mutants have been created by applying muJava to three selected single-mutation mutants, each of which has a different mutation operator used by muJava in creating mutants. The three selected operators are COR, ROR, and AOR. Note that each one of the three selected single-mutation mutants does not have any independent bad pair. That is, they are part of the 46 single-mutation mutants that do not have any independent bad pair. There are 250 double-mutation mutants for each set and the same set of test cases are executed repeatedly on these mutants.

Figure 5.5 shows the summary plot from the test run on the COR set of 250 double-mutation mutants. There are 3 mutants that have 11 independent bad pairs; 37 mutants have at least 17 independent bad pairs.

Figure 5.6 shows the summary plot from the test run on the ROR set of 250

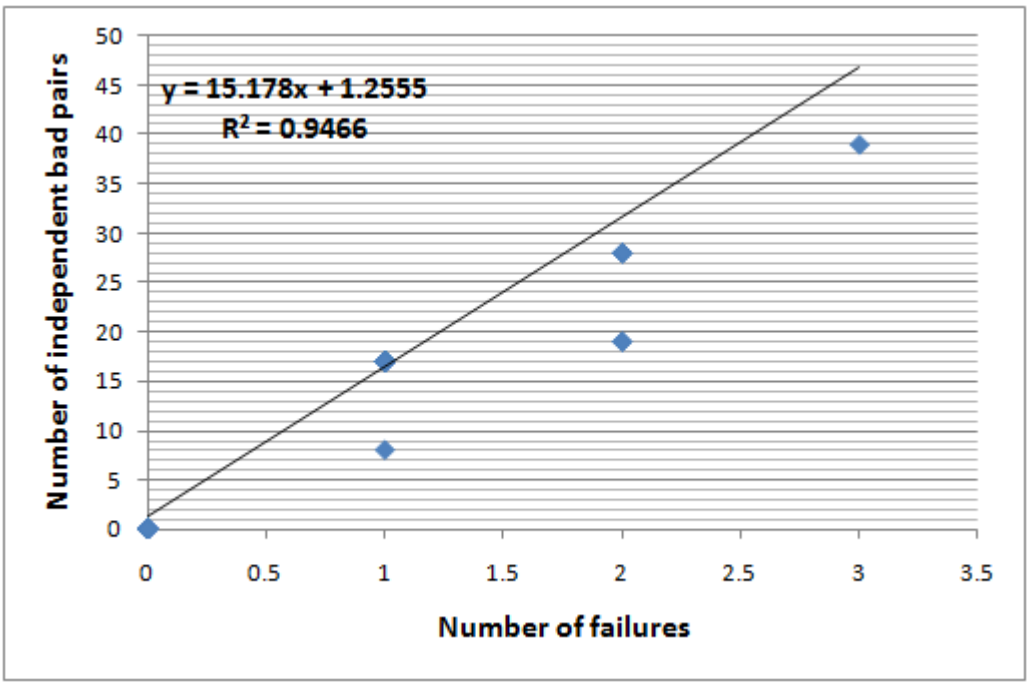

Figure 5.4: A positive relationship exists between the number of failures and the number of independent bad pairs with R-square correlation coefficient equal to 0.9466 based on 244 data points.

double-mutation mutants. There are 8 mutants that have 8 independent bad pairs; 20 mutants have 17 independent bad pairs.

Figure 5.7 shows the summary plot from the test run on the AOR set of 250 double-mutation mutants. There are 3 mutants that have 8 independent bad pairs; 22 mutants have at least 17 independent bad pairs.

5.4 Discussion

Several observations are worth noting from the case study. In testing mutants with a single seeded fault, independent bad pairs are common because 204 out of the 250 mutants have at least 8 independent bad pairs. Changing the threshold of failure ratio from 1.0 to 0.6 does not affect the independent bad pairs observed, but at threshold 0.5 the number of independent bad pairs more than doubles. By lowering the thresholds for identifying bad pairs, one can locate nearly bad pairs. This can be a useful alternative when no bad pairs exist at the default failure ratio of 1.0. Moreover, there is a positive relationship between the number of failed test cases and the number of independent bad pairs with R-square correlation coefficient equal to 0.9466.

In comparing single mutation and double mutation, the number of mutants with no independent bad pairs decreases significantly when mutants contain two seeded faults. Further, the overall number of independent bad pairs increases when mutants contain two seeded faults. This suggests that a potential relationship exists between the number of seeded faults and the number of independent bad pairs.

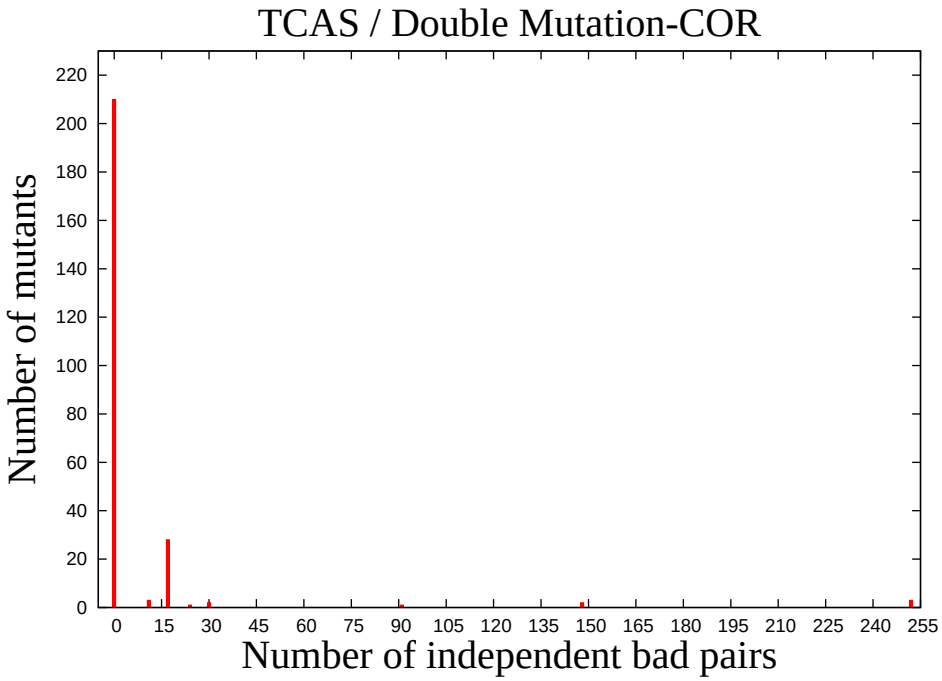

Figure 5.5: The 250 double-mutation mutants are created from a COR single-mutation mutant that does not have any independent bad pair. The analysis of bad pairs show that 3 out of the 250 double-mutation mutants have 11 independent bad pairs; 28 have 17 independent bad pairs; 1 has 24 independent bad pairs; 2 have 30 independent bad pairs; 1 has 91 independent bad pairs; 2 have 148 independent bad pairs; 3 have 252 independent bad pairs.

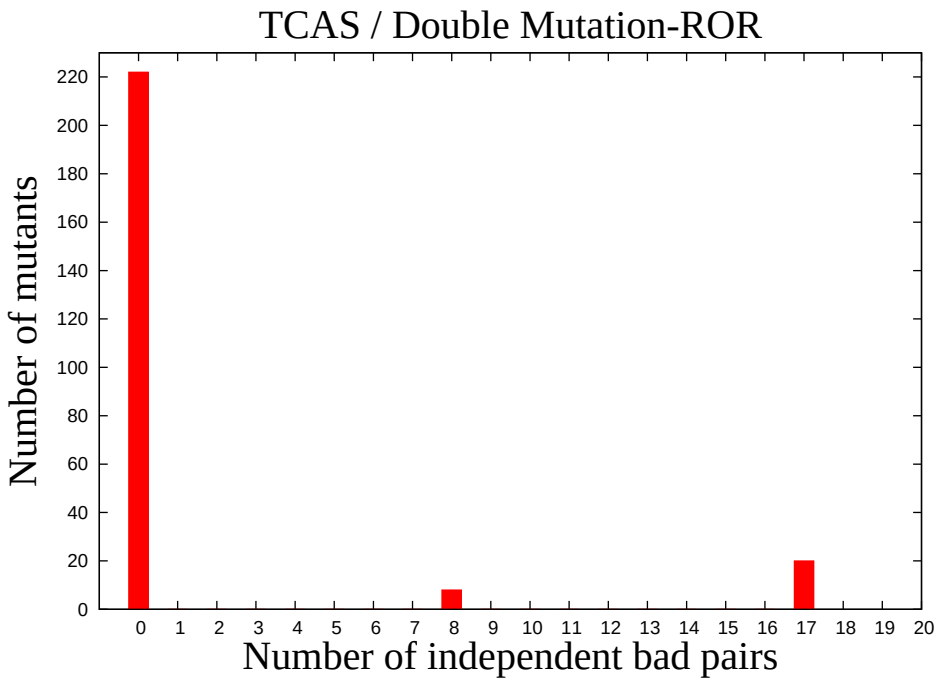

Figure 5.6: The 250 double-mutation mutants are created from a ROR single-mutation mutant that does not have any independent bad pair. The analysis of bad pairs show that 8 out of the 250 double-mutation mutants have 8 independent bad pairs, and 20 have 17 independent bad pairs.

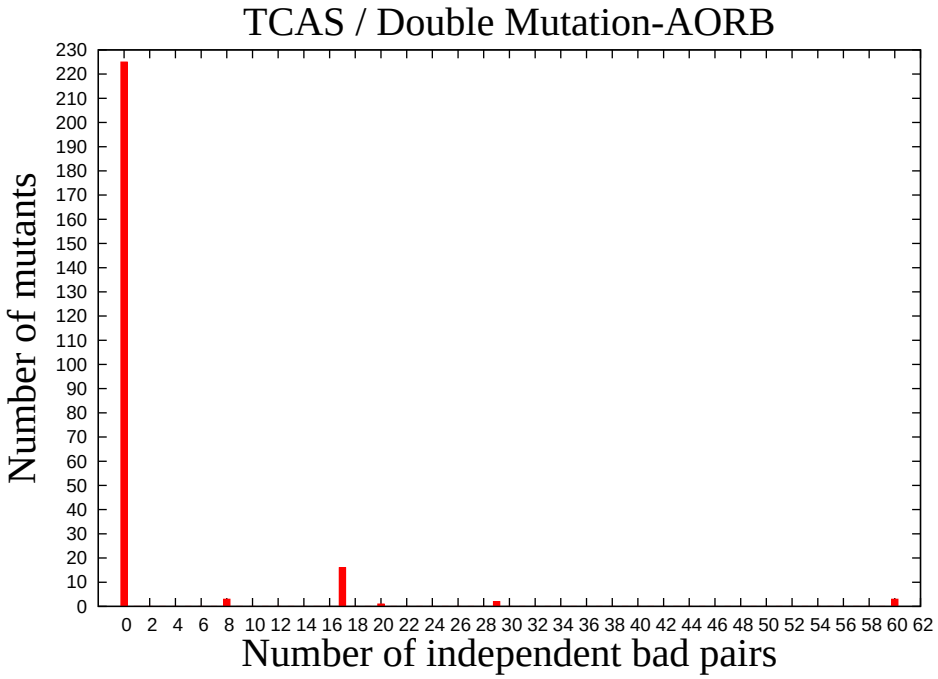

Figure 5.7: The 250 double-mutation mutants are created from a AOR single-mutation mutant that does not have any independent bad pair. The analysis of bad pairs show that 3 out of the 250 double-mutation mutants have 8 independent bad pairs; 16 have 17 independent bad pairs. 1 have 20 independent bad pairs; 2 have 29 independent bad pairs; 3 have 60 independent bad pairs.

Chapter 6
Case Study: Bad Pairs in Network Vulnerability Testing

In this case study, BadPair is used to identify bad pairs by analyzing a test table generated by the Achilles Satellite, developed by Wurldtech Security Technologies Inc. [3], in network vulnerability testing. The purpose of the case study is twofold. It shows the application of BadPair to industrial application software. In addition, it demonstrates the scenario of using BadPair given only a test table in the absence of target source code, as shown in Figure 6.1

The device-under-test (DUT) was a network device, and the Achilles Satellite was used to test the network device by sending large number of network packets to it. Each packet represented one test case. In one particular test batch there were 6734 network packets transmitted to the network device. To use BadPair to identify the bad pairs from the 6734 packets, the same 11 IP header fields were extracted from each packet. These 11 header fields were selected because they were determined to be viable parameters. Table 6.1 shows the possible values extracted from the 6734 packets for the 11 header fields. Before using BadPair to identify the bad pairs, manual inspection of the given test table revealed that 2 packets, packet 4188 and packet 4220, out of the 6734 packets, caused the DUT to fail the test. Nevertheless, the specific characteristics of the input parameters in the two failed packets remained unclear.

6.1 Bad Pairs Analysis

Given the test table that contains the test results from executing the 6734 packets, manual analysis of the test table of 6734 packets would be an extremely challenging task. Instead, BadPair was applied to perform bad pairs analysis. Figures 6.2 and 6.3 show the respective execution flow and pseudocode in applying BadPair to identify

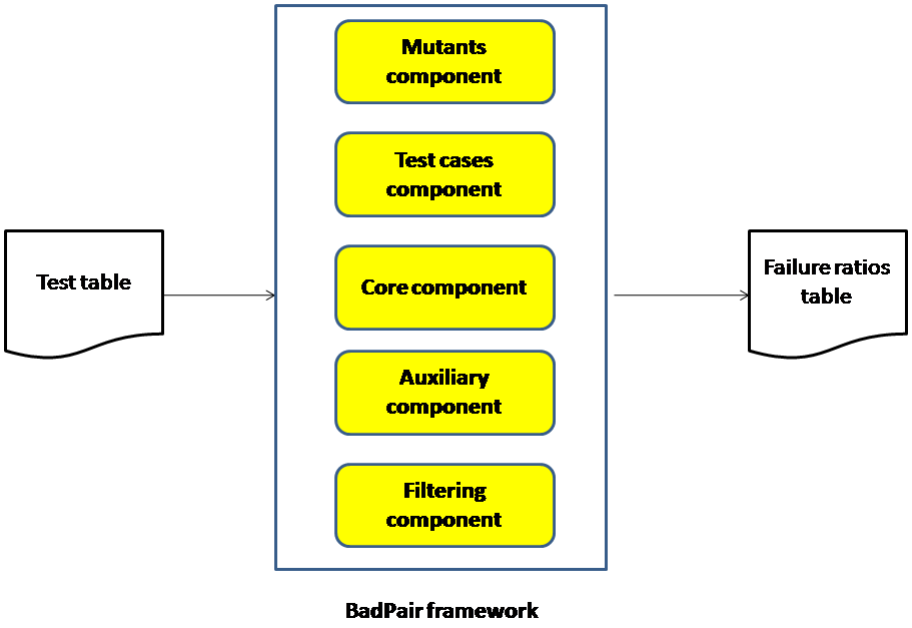

Figure 6.1: Invoking the BadPair framework given only a test table

Version	header length	TOS	total length	id	flags	offset	TTL	protocol	source	destination
0	0	0	0	0	0	0	0	0	0x00000000	0x00000000
1	4	1	19	1	1	1	1	1	0xc0a86401	0xc0a8644d
4	5	3	20	65534	2	1479	2	2	0xc0a8644d	0xc0a86401
6	6	5	1499	65535	4	1480	3	3	0xc0a864ff	0xc0a864ff
7	14	7	1500		7	1481	31	5	0xefffffff	0xefffffff
	15		1501		3	1499	32	6	0xffffffff	0xffffffff
			3001		5	1500	33	7		
			65534		6	1501	63	16		
			65535			8190	64	17		
						8191	65	18		
							127	37		
							128	59		
							129	64		
							254	65		
							255	254		
								255		

Table 6.1: IP parameters

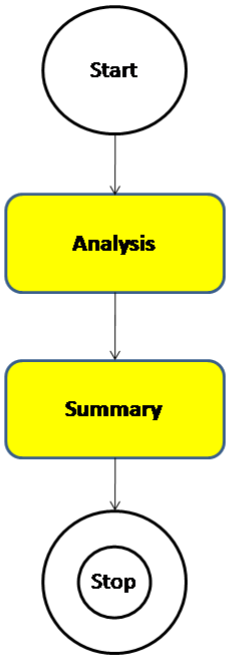

Figure 6.2: BadPair execution flow of the case study in network vulnerability testing

bad pairs for this case study. There are 55 parameter pairs that can be derived from each failed packet, and all are candidate bad pairs. (There are 55 pairs because each packet consists of 11 parameter values and ${}_{11}C_2 = 55$.) The two failed packets appeared in the test table as follows:

```
0 5 0 0 0 0 0 64 1 c0a86401 c0a8644d F
0 5 0 0 0 0 0 64 6 c0a86401 c0a8644d F
```

6.2 Analysis Results

The result from bad pairs analysis points out that there are two bad pairs and both are independent bad pairs:

$(\bullet, \bullet, \bullet, 0, \bullet, \bullet, \bullet, \bullet, 6, \bullet, \bullet)$

$(\bullet, \bullet, \bullet, 0, \bullet, \bullet, \bullet, \bullet, 1, \bullet, \bullet)$

That is, whenever the total length equals 0 and the protocol equals 1, the DUT failed the test. Similarly, whenever the total length equals 0 and the protocol equals 6 , the

open test table log file L
open failure ratios table log file F
generate failure ratios from L
write failure ratios to F
close F
close L

Figure 6.3: BadPair execution pseudocode of the case study in network vulnerability testing

DUT failed the test.

6.3 Discussion

Given the test table that contains the results of executing 6734 test cases on testing a network device, we were able to use BadPair to pinpoint the two independent bad pairs that always result in failures. We believe this is valuable information in facilitating the debugging process.

Chapter 7
Related Work

7.1 Software Testing Is Costly

Manual inspection of complex source code is time-consuming and error-prone [13]. Automated software testing is one way to cope with software quality assurance. However, testing alone is not enough to guarantee that the System-Under-Test (SUT) is free of bugs. Nonetheless, it adds confidence in the correctness of the SUT. Failures of software resulting from combinations of input parameters have long been studied and researched [16]. Testing every possible combination of the parameter values is often impractical and too expensive to execute [5, 18]. The sheer volume of combinations of multiple parameter values alone presents a challenging task in the debugging process in software testing. Instead, one remedy is to use n-wise testing where a subset of all possible combinations is used instead as various studies suggest that it can be an effective or practical option [5, 7]. However, the number of combinations to test on the SUT can remain huge when the number of parameters and the number of parameter values are large. A common approach is to focus on failures caused by the combinations of two parameters [24, 10, 16, 9].

7.2 N-wise Testing

Cohen et al. [8, 9] at Bellcore created a tool that allowed subsets of all possible combinations of parameters to be generated based on user-specified requirements. The tool was evaluated in several experiments at Bellcore. It was concluded that substantial savings in testing were achieved by using the tool.

Lei el at. presented FireEye, a testing tool using their proposed n-wise testing strategy. Several experiments were conducted to evaluate FireEye. The authors concluded that FireEye performed significantly superior than other testing tools in terms of the execution time and the number of test cases.

7.3 Pairwise Testing

Various studies suggest that pairwise testing can be an effective or practical option. Based on the analysis of data from defected medical devices over 15 years, Wallace et al. [24] suggested that pairwise testing could detect about 98% of device failures. Using four case studies, Dalal et al. [10] showed the effectiveness of pairwise testing. Kuhn and Reilly [16] investigated the bug tracking reports of open source software, and concluded that about 70% of failures were related to more than two conditions. Cohen el al. [9] at Bellcore found that most field errors were caused by combinations of at most two parameter values based on empirical results.

Though pairwise testing can be effective, it is not perfect. Moritz [21] identified the Minimum Conditions required to cause failures and found in a case study that almost one third of failures required at least 3 conditions to trigger them. Thus, a blind reduction of the number of test cases in pursuit of a minimum set of pairwise test cases could be risky.

7.4 Mutation Testing

One approach to gauging the quality of software is by mutation testing. Seeded faults are injected intentionally in the SUT to create mutants. Then, the test suite is executed once on each mutant. Afterwards, mutation analysis focuses on the ratio of killed mutants, which is used as an indicator for the software quality.

Ma et al. designed and implemented a mutation analysis tool, muJava [19, 2], targeted at Java programs. muJava is capable of generating a high volume of mutants by manipulating mathematical operators in source code. It can also determine software quality by calculating the percentage of killed mutants. Haschemi and Weibleder [11] proposed a framework for mutation analysis where the framework served as a bridge between mutation analysis tools and execution environments. A prototype of the framework was implemented and integrated with two mutation analysis tools as case studies. The framework defined an application programming interface, which each mutation tool had to implement to be successfully integrated into the framework. In addition, the framework separated mutant creation and test execution to support test

run on two different execution environments.

7.5 Error Locating Arrays

Suppose "safe values" are those parameter values that do not cause the SUT to fail. Stevens et al. [20] proposed Error Locating Arrays (ELA). Assuming the safe values for the SUT are known in advance, an ELA can then be used to locate failures caused by combinations of multiple parameters. For instance, in its most generalized format, an ELA is an input table where every n-wise parameter combination must be contained in at least one row of the input table. In addition, each row contains a n-tuple that is the only n-wise combination in the row that causes the SUT to fail.

Chapter 8
Conclusions

In this thesis, a novel approach to analyzing test outputs is proposed. Our approach focuses on bad pairs by analyzing test outputs to identify the pairs of input conditions that always result in failures on the tested system. To validate and illustrate the ideas, the BadPair framework has been implemented. In addition, three case studies have been conducted using the BadPair framework. The first two case studies provide insights into the relationships between bad pairs, test cases, and faults. In the last case study, the BadPair framework is applied to an industrial network device in network vulnerability testing.

We believe the thesis has made several contributions to the research in software testing. First, to the best of our knowledge, the bad pairs analysis is a relatively novel approach as little related research effort is known. Second, the proposed approach has been modeled and implemented in the BadPair framework presented together with three case studies. Third, the results from the case studies conclude that there is a positive relationship between the number of independent bad pairs and the number of failed test cases. Fourth, the results also suggest that there is a positive relationship between the number of independent bad pairs and the number of faults. In addition, filtering of test cases has a significant impact on the bad pairs identified. Furthermore, changing the threshold of failure ratio does not seem to have much impact on the bad pairs. Lastly, the last case study demonstrates how the BadPair framework can be utilized to facilitate the debugging process in testing an industrial network device.

Chapter 9
Future Work

Possible expansion of this thesis includes, but not limited to, the following:

- Bad triplets and quadruplets
- Other case studies
- Improvements on the BadPair framework

9.1 Bad Triplets and Quadruplets

In the case studies presented in Chapters 4 and 5, the thresholds of failure ratios were lowered in search of 'nearly' bad pairs. These nearly bad pairs may lead to discovery of bad triplets or quadruplets. For instance, the test table shown in Figure 9.1 contains the pair, $(3, 4, \bullet, \bullet)$, which causes failures 75% of the time. Further inspection of the test table reveals that the triplet $(3, 4, 4, \bullet)$ results in failures 100% of the time and it is in fact a bad triplet. Therefore, it can be valuable to expand the BadPair framework to identify bad triplets and even bad quadruplets.

9.2 Other Case Studies

The three case studies conducted in this thesis have validated the correctness and demonstrated the usefulness of the bad pairs analysis. Nevertheless, case studies on more diversified software programs can serve to complement the existing case studies and to further improve the BadPair framework.

9.3 Improvements on the BadPair framework

Currently, log files are used to store intermediate results from test run and to store results from analysis of bad pairs. Alternatively, a database engine can be incorporated to store the resulting data. This will facilitate the extracting and retrieving

3	4	5	5	P
3	4	4	6	F
3	4	4	7	F
3	4	4	8	F

Figure 9.1: Example of a bad triplet: $(3, 4, 4, \bullet)$

of information from the data given the robustness and power of database engines, particularly when the data is of huge volume and complexity. Also, there is no GUI (graphical user interface) provided in the current release of the BadPair framework. An addition of a GUI tool will certainly make the BadPair framework more user friendly and appealing.

Bibliography

[1] Siemens programs. `http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/`.

[2] ujava home page. `http://cs.gmu.edu/~offutt/mujava/`.

[3] Wurldtech home page. `http://www.wurldtech.com/`.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge, 2008.

[5] K. Z. Bell and M. A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *Information and Communications Technology. ITI 3rd International Conference on Enabling Technologies for the New Knowledge Society*, pages 221–235, May-Jun 2005.

[6] M. C. Chang, D. Hoffman, D. Bazdell, and K. Yoo. Bad pairs in software testing. To appear in TAIC-PART, IEEE Computer Society, 2010.

[7] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, Jul 1997.

[8] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton. The automatic efficient test generator (AETG) system. In *Proceedings of The 5th International Symposium on Software Reliability Engineering*, pages 303–309, Jun 1994.

[9] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *International Symposium on Software Reliability Engineering, IEEE Computer Society*, 13(5):83–88, Sep 1996.

[10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. *International Conference on Software Engineering*, 0:285–294, 1999.

[11] S. Haschemi and S. Weibleder. A generic approach to run mutation analysis. To appear in TAIC-PART, IEEE Computer Society, 2010.

[12] D. Hoffman, D. Ly-Gagnon, P. Strooper, and H. Wang. Grammar-based test generation with YouGen. To appear in Software: Practice and Experience, Wiley InterScience, 2010.

[13] D. Hoffman and P. Strooper. *Software Design, Automated Testing, and Maintenance*. International Thomson Computer Press, 1995.

[14] Daniel Hoffman, Hong-Yi Wang, Mitch Chang, and David Ly-Gagnon. Grammar based testing of HTML injection vulnerabilities in RSS feeds. *TAIC-PART, IEEE Computer Society*, 0:105–110, Sep 2009.

[15] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering, IEEE Computer Society*, 24(10):779–796, 1998.

[16] D.R. Kuhn, D.R. Wallace, and Jr. Gallo, A.M. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, Jun 2004.

[17] W. Langdon, M. Harman, and Y. Jia. Multi objective higher order mutation testing with genetic programming. *TAIC-PART, IEEE Computer Society*, 0:21–29, Sep 2009.

[18] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, IEEE Computer Society*, pages 549–556, 2007.

[19] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system: Research articles. *Software Testing Verification Reliability*, 15(2):97–133, 2005.

[20] Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM J. Discrete Math.*, 23(4):1776–1799, 2009/10.

[21] E. Moritz. Case study: how analysis of customer found defects can be used by system test to improve quality. In *Proceedings of the International Conference on Software Engineering*, pages 16–24, 2009.

[22] L. Sobotkiewicz. A new tool for grammar-based test case generation. Master's thesis, Univ. of Victoria, Dec 2008.

[23] M. Stoelinga and M. Timmer. Interpreting a successful testing process: Risk and actual coverage. In *Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 251 –258, 2009.

[24] Dolores R. Wallace and D. Richard Kuhn. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality & Safety Engineering*, 8(4):351, 2001.