Comparing Ochiai and Relief for Spectrum-based Fault Localization

Brian Oburak Omoro

Comparing Ochiai and Relief for Spectrum-based Fault Localization

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Brian Oburak Omoro
born in Torit, South Sudan

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Comparing Ochiai and Relief for Spectrum-based Fault Localization

Author: Brian Oburak Omoro
Student id: 4025687
Email: b.o.omoro@student.tudelft.n

Abstract

Fault localization is one of the activities of system diagnosis and its goal is to pinpoint the precise locations of faults in systems. This process is recognized as one of the most tedious, time-consuming and expensive undertakings of fault diagnosis. Consequently research in this domain have lead to the birth of numerous approaches to automate the process in order to minimize failures and produce reliable systems. Among the proposed fault localization approaches are the statistical-based Spectrum-based Fault Localization (SFL) and machine learning based Feature Selection Relief. In SFL, the assumed faultiness of a system component is computed using a similarity coefficient and the most commonly used coefficients are Ochiai, Tarantula and Jaccard. Currently, Ochiai clearly outperforms most of the known similarity coefficients in SFL. The Feature Selection based Relief, in short known as Relief, is an alternative technique that has been recently proposed for fault localization. The Relief technique works by assigning relevance weights to components and the components that are likely to be faulty receive the highest relevance weights. In this document, we describe the study performed to compare the performance of Ochiai and Relief for SFL in various systems using the SFL-Simulator which is a Ruby-based tool used for research in SFL. Results from the study indicate that the diagnostic performance of both fault localization methods largely depends on the configuration of the system under investigation, i.e. the number of faults, the health states of the faulty components, the constituent components and the links between them and the number of transactions or test runs used. Furthermore, the study has shown that Ochiai has a computational complexity that is superior to Relief.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Hans-Gerhard Gross, Faculty EEMCS, TU Delft
Committee Member:	Dr. Claudia Hauff, Faculty EEMCS, TU Delft

Preface

This Master's thesis written in partial fulfillment of the requirements for the degree of the Master of Science in Computer Science, is a result of a research which I conducted at the Software Engineering Research Group (SERG) at Delft University of Technology. I had an interesting opportunity to perform a comparative study on software fault localization techniques. It is conceivably a huge acquisition of Software Engineering knowledge for me, an indispensable contribution to my career as a software engineer.

I would first like to thank my supervisor, Dr. Hans-Gerhard Gross, for giving me the opportunity to carry out this research under his wings. I am thankful for his continuous support and invaluable technical assistance. I would additionally like to express my gratitude to Cuiting Chen, for her inputs and discussions during those few brief meetings we had together with her and Dr. Hans-Gerhard Gross.

I would also like to thank my friend, Joey Siadis for the chats, help, advice and endless inputs during the whole research duration. Last but not least, I would like to thank my parents, Dr. Elijo Omoro and Marcelina Ikikila for instilling in me the belief of how important education is in life and their endless support and love, without which it would be humongously hard for me to get where I am standing right now.

Brian Oburak Omoro
Delft, the Netherlands
July 18, 2013

Contents

List of Figures

Chapter 1

Introduction

There are many reasons why faults exist in software systems. Although a system is consistently tested during the development phase, faults may still persist in the system and eventually emerge when the system is in production [3]. Additionally, systems are continuously altered over their lifetime during maintenances and upgrades. These activities can contribute to the introduction of additional faults into the system. In order to resolve these faults, it is essential to pinpoint where they reside. By definition a *fault* is what elicits an *error* in the system, and an error defines a system state that may evolve into a *failure* which is an event that may result in the delivery of improper services that deviate from the correct or expected services [4]. Therefore, in order to prevent incorrect delivery of services due to faults, it is important to locate and diagnose these faults before they evolve into errors and eventually into failures. Fault localization is one of the activities of system diagnosis[1] and its goal is to pinpoint the precise locations of faults in systems [2].

For the past few years a number of promising fault localization techniques have been proposed. Most of these techniques pertain predominantly to two categories namely the *statistical* and *machine learning* categories [5]. Generally, the statistical approaches find faults in the system using abstraction of program traces known as *program spectra* collected at runtime, whereas the machine learning approaches use mainly a static model of the behavior of the system. However, some approaches from the machine learning domain have taken the best-of-two-worlds approach and like their statistical counterparts, exploit program spectra exclusively [6] or in combination with system models [7] to detect fault locations in the system.

Spectrum-based Fault Localization (SFL) techniques are a class of statistical fault localization techniques that are known to be among the best statistical approaches because of their low-cost diagnosis and accuracy [8]. They utilize system failure behavior and activity collected as program spectra to predict the likelihood of faultiness of constituent system components [9]. In SFL, the assumed faultiness of a system component is computed using a *similarity coefficient* and the most commonly used coefficients are *Ochiai* [2], *Tarantula* [1] and *Jaccard* [8]. Currently, Ochiai undoubtedly outperforms most of the known similarity coefficients in SFL [4].

[1] Diagnosis and debugging are synonymous.

Machine learning approaches, conventionally diagnose a system by reasoning over its model. Consequently, although mostly accurate than their SFL counterparts, they are more complex. Nonetheless some machine learning techniques have taken advantage of program spectra like in SFL to locate faults with a reasonable complexity. Some of these are the Bayesian-based BARINEL [10] and Feature Selection based Relief [6] techniques. Relief is less complex than BARINEL because it exclusively uses the program spectra to locate faults without the need for a system model. The Relief technique works by assigning relevance weights to components and the components that are likely to be faulty receive the highest relevance weights.

Fault localization techniques have a common denominator namely *accuracy*. The accuracy of a technique is a measure of its performance or effectiveness to correctly identify the locations of faults in a system. Consequently, the accuracy of a technique is one of the main factors to consider when making a choice between several fault localization techniques. In this thesis we investigate SFL using the Ochiai[2] and Relief[3] techniques and the extent to which these two techniques are comparable to one another. The goal of this thesis is to gain insight in the effectiveness of the two methods using a number of different systems.

1.1 Problem Statement

Contemporary software systems have an ever increasing complexity and their round-the-clock availability to users has become a necessity. Diagnosing these systems and detecting the locations of faults is a daunting task. Hence, the last few years have seen a spurt of state-of-the-art automatic diagnosis techniques in the domain of fault localization to minimize failures and produce reliable systems. Two such fault localization techniques are Ochiai and Relief.

In numerous previous studies Ochiai has been proven to be among the best SFL techniques specially in terms of fault localization accuracy. Abreu et al. [4] in their article about accuracy of spectrum-based fault localization presented results that demonstrated Ochiai having an improved diagnostic accuracy over other eight SFL methods. In another article by Abreu et al. [8] results showed repeatedly that Ochiai outperforms Tarantula and Jaccard. The toolset talked about in the study by Janssem et al. [11] also produced diagnostic results that illustrate that Ochiai is better than its counterparts. Naish et al. [12] used several coefficients to evaluate their SFL approach and for single-fault programs their results demonstrated that Ochiai performs better than the rest. From the results obtained in these studies and others it can be affirmed with some degree of certitude that Ochiai is a reliable and feasible SFL technique.

For fault localization methods stemming from machine learning that use exclusively program spectra as input for fault localization diagnosis, there have been a very few studies and among them is the article by Roychowdhury et al. [6] in which they proposed the Relief method for locating faults in systems. Their experimental results exhibited that using the Siemens Test Suite, Relief performs similarly to Ochiai. However, according to the au-

[2]Ochiai is analogous with SFL using the Ochiai similarity coefficient.

[3]Relief is analogous with fault localization using the Feature Selection based Relief Algorithm.

thors by using more code coverage Relief starts to perform better than Ochiai. They stated that a merely 20% code coverage uncovers 90% of faults and 95% at 30% code coverage compared to 88% faults discovered by Ochiai under the same code coverage.

The Siemens Test Suite has been used extensively in the majority of SFL research studies and among them those mentioned above. This test suite is feasible for studying programs at a low component granularity level namely at the level of functions and statements. However, for bigger systems such as service-oriented systems, a larger grain size for components is more befitting because it reduces monitoring and performance overhead [2]. One way to do this is to model systems into component topologies with larger component granularity. The main advantage of modeling the systems into topologies is that it allows us to easily simulate the system and we can run thousands of system simulations within a reasonable time. Additionally, using a system topology model of the base system, many variables of the system can be altered easily to generalize findings according to different configurations of the system and this averts system implementation details that may negatively affect observations. A very few studies have explicitly employed system topologies for their SFL studies [13] and none have ever studied the impact of Ochiai and Relief on different systems.

The goal of this thesis is to investigate how Ochiai and Relief methods will perform on different systems with larger component granularity. The SFL-Simulator [14] will largely be used to simulate the systems under investigation. It is of paramount importance to find out in which situations one fault localization method performs better than the other to enable employing the right method in a specific scenario. The problems at hand give rise to a number of research questions that need to be addressed. The goal of this thesis is to provide answers to the following two main questions:

- **Research Question 1**:
  Considering the two techniques Ochiai and Relief: Which one has a better fault localization performance or accuracy?

- **Research Question 2**:
  Are there characteristics of systems that support the application of: Ochiai or Relief?

1.2 Contributions

We identify the following contributions of this thesis:

1. Identifying the characteristics of systems that affect fault localization performance (Chapter 4).

2. Performing numerous Ochiai/Relief fault localization experiments with different systems (Chapter 4 and 5).

3. Proposed new performance metric for measuring the diagnosis accuracy or effectiveness of a fault localization method (Chapter 4).

4. Assessing and evaluating the effectiveness of Ochiai and Relief for different systems based on the experimental results (Chapter 6).

1.3 Thesis Structure

The thesis is structured as follows. Chapter 2 comprises background information about SFL and Relief. The SFL-Simulator tool is presented in Chapter 3. We describe the experimental setup in chapter 4 and experimental results in chapter 5. The elaboration of the experimental results is in Chapter 6, with the thread to validity of the results in Chapter 7. Related work about Ochiai and Relief is in chapter 8. Finally, conclusions and future work are presented in Chapter 9.

Chapter 2

Background

This chapter provides some background information on the topics discussed in this thesis. Section 2.1 begins with concepts and terminology followed by Section 2.2 which presents an understanding about SFL and the Ochiai method, whereas Section 2.3 provides a global introduction to the Feature Selection based Relief fault localization method.

2.1 Concepts and Terminology

2.1.1 Fault, Error and Failure

A *failure* is an event that occurs when the service delivered deviates from the correct or expected service, an *error* is a system state that may result in a failure and a *fault*, also know as a *bug*, is the cause of an error in the system [4].

```
1  mid(){
2      int x, y, z, m;
3      read("Enter 3 numbers:", x, y, z);
4      m = z;
5      if (y < z){
6          if(x < y){
7              m = y;
8          }else if (x < z){
9              m = y;  // fault location
10         }
11     }else{
12         if (x > y){
13             m = y;
14         }else if (x > z){
15             m = x;
16         }
17     }
18     print("Middle number is:", m);
19 }
```

Figure 2.1: A defective C program [1].

To put the aforementioned concepts into perspective, lets consider Figure 2.1 used in the article by Jones et al. [1] which depicts a faulty C program. The logic of the function *mid()* is as follows, it takes three integer inputs and returns their median. The function contains a fault on line 9 at the statement $m = y;$. For all values of $x < y < z$, the function *mid()* will work properly. The problem starts when we have a situation where $y < x < z$, in this case x should be denoted as the median but due to the fault in line 9 the variable y will be designated as median instead. This is a fault which can propagate into an error and eventually into a failure with an incorrect output as a result. This example shows clearly that faults are the cause of failures but they do not necessarily lead to errors and failures as long as the faulty conditions or paths are not executed. Nonetheless, it is crucial to uncover them before they cause problems in the system.

2.1.2 Transaction

A *transaction* or an observation or a test run marks the start and end of a single execution activity of a system. During this activity depending on the path taken a number of components may or may not be involved (covered or activated) during the execution.

2.1.3 Program Spectrum and Error Vector

A *program spectrum* is a collection of data that projects the execution activity of a system, denoting which program components are covered (active) or not covered (inactive) in particular transactions. The program components can be entities such as software systems, source code classes, functions, statements, branches, paths or basic blocks. The components are allocated binary values (1 or 0), also known as counters or flags. A ”1” is assigned to a component to signify whether it has been covered and a ”0” if it has not been covered during a particular transaction [2, 9]. Besides the program spectrum, another important piece of information known as the *error vector* is obtained during transactions. The error vector holds binary values representing the outcomes of individual transactions[1]. The error vector denotes failed and passed transactions by ’1’ and ’0’ respectively. Both of these pieces of information are vital for the fault localization process because potential faulty components can be identified by computing the similarity between the program spectra and the error vectors. There are many different kinds of program spectra [15], see Table 2.1. The most commonly used program spectra type is the block hit program spectra. In a block hit spectrum each component is represented by a flag or a counter to indicate whether the block of code or statement or component was executed during a particular transaction.

2.1.4 Similarity Coefficient (SC)

In SFL, the assumed faultiness of a component is computed using a *similarity coefficient* (SC). Thus, the SC is a function that quantifies the similarity between the program spectra of components and the error vectors to rank components based on their likelihood to contain

[1] Test runs and transactions are synonymous

	Name	Description
BHS	Branch Hit Spectra	conditional branches that are executed
BCS	Branch Count Spectra	number of times each conditional branch is executed
CPS	Complete Path Spectra	complete path that is executed
PHS	Path Hit Spectra	loop-free path that is executed
PCS	Path Count Spectra	number of times each loop-free path is executed
DHS	Data-Dependence Hit Spectra	definition-use pairs that are executed
DCS	Data-Dependence Count Spectra	number of times each definition-use pair is executed
OPS	Output Spectra	output that is produced
ETS	Execution Trace Spectra	execution trace that is produced
DVS	Data Value Spectra	the values of variables in the execution
ESHS	Executable Statement Hit Spectra	executable statements that are executed

Table 2.1: Types of Program Spectra [15].

faults. Many SC formulas exit, but the most commonly used SCs in SFL are *Ochiai* [2], *Tarantula* [1] and *Jaccard* [8].

2.2 Spectrum-based Fault Localization (SFL)

SFL uses program spectra and error vectors as input data to localize faulty components in a system. SFL can be performed online and offline and both versions utilize the same input information. The two versions differ only in the way how they run tests or transactions. Unlike its offline counterpart, online SFL is applied to a system that is executing. Due to the continuous execution nature of such a system the *active testing* used in the offline version [9] is not efficient because it will cause undesirable interference with the system. Thus, in online SFL the transaction and its validation must incorporate the continuous nature of the system to reduce interference with the functions of the system. The functioning of the system is overseen by monitors which report on the proper execution of the system by providing test or transaction outcomes [2]. Monitoring is more fitting in online SFL because of its passive nature, for example, it can be triggered by an event such as the arrival of new data or a timer interrupt. A monitor is a specific component in the system that observes and assesses the validity of the functionality without interfering with the rest of the system. It oversees data or behavior of the system at specific locations and decides based on built-in oracle logic whether an observation is expected (*pass*) or unexpected (*fail*). The observation can occur, for instance, by checking the range of a variable, consistency between different data, or through comparison with a state model. The monitor outcomes are what test outcomes are for offline SFL [2]. SFL takes in the input information to yield an ordered ranking of the program components based on their likelihood or suspicion to be faulty.

2.2.1 Input Data

SFL takes the following data as input [9]:

- A finite set $C = \{c_1, c_2, ..., c_j, ..., c_M\}$ of M components of a program P.
- A finite set $T = \{t_1, t_2, ..., t_i, ..., t_N\}$ of N tests (transactions)
- A finite set $O = \{o_1, o_2, ..., o_i, ...o_N\}$ of binary outcomes for the set T with $o_i = 0$ if test t_i passed and $o_i = 1$ otherwise. The set O represents the error vector e (see Figure 2.2).
- $A[a_{ij}]$ is an M x N matrix known as the *coverage* or *activity* matrix, where $a_{ij} = 1$ if test or transaction t_i covers component c_j, and $a_{ij} = 0$ otherwise. Each row vector R in the matrix, with $R = \{A[i] \mid 0 \leq i < M\}$, is known as a *spectrum* (see Figure 2.2).

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{bmatrix} e = \begin{bmatrix} e_1 \\ e_2 \\ \cdots \\ \vdots \\ e_N \end{bmatrix}$$

Figure 2.2: Input for SFL

2.2.2 Test Runs Data

As depicted in Figure 2.2, the activity matrix A comprises N test runs and M different program components. The information about fail/pass outcomes of runs constitutes the vector column e, which is the error vector. For each program component j and test run i, the frequency of runs is defined by the equation [16]:

$$n_{pq}(j) = |\{i | a_{ij} = p \wedge o_i = q\}| p, q \in \{0, 1\}, \tag{2.1}$$

where p is 1 if the component j is covered, and 0 otherwise. On the other hand, q is 1 if test i fails, and 0 otherwise. For example, Table 2.2 contains all possible a_{ij} counters for a particular component, and for each component these counters constitute the number of test runs N.

Component	Passed Test Run	Fail Test Run
Covered	a_{10}	a_{11}
Not covered	a_{00}	a_{01}

Table 2.2: Different types of test run frequency count.

2.2.3 Diagnosis Computation

The diagnosis score D_j for each component j is calculated using the information from Table 2.2 as input for the similarity coefficient method. Underneath are three commonly used SCs namely Ochiai, Tarantula and Jaccard:

$$Ochiai: \ D_j = \frac{n_{11}(j)}{\sqrt{(n_{11}(j)+n_{01}(j)).(n_{11}(j)+n_{10}(j))}} \tag{2.2}$$

$$Tarantula: \ D_j = \frac{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)}}{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)} + \frac{n_{10}(j)}{n_{10}(j)+n_{00}(j)}} \tag{2.3}$$

$$Jaccard: \ D_j = \frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)+n_{10}(j)} \tag{2.4}$$

where $n_{11}(j)$ is the number of failed transactions in which component j is involved, $n_{10}(j)$ is the number of passed transactions in which component j is covered, $n_{01}(j)$ is the number of failed transactions where component j is not involved and $n_{00}(j)$ is the number of passed transactions in which component j is not involved. Using Figure 2.2 and Table 2.2 this can be summarized as follows [10]:

$$n_{00}(j) = |\{i|a_{ij} = 0 \wedge e_i = 0\}|, \tag{2.5}$$

$$n_{01}(j) = |\{i|a_{ij} = 0 \wedge e_i = 1\}|, \tag{2.6}$$

$$n_{10}(j) = |\{i|a_{ij} = 1 \wedge e_i = 0\}|, \tag{2.7}$$

$$n_{11}(j) = |\{i|a_{ij} = 1 \wedge e_i = 1\}| \tag{2.8}$$

The ranking D_j for each component is computed by comparing row vector or spectrum of the component from the coverage matrix with the error vector to produce its likelihood to be at fault.

2.2.4 Example of SFL Diagnosis

To illustrate the application of SFL on a sample of code, the example from [2] which is based on a C function is used. The components in this example are denoted by the statements of the function as can be seen in Figure 2.3. The function counts different types of characters and contains a fault at component c_3 because it mishandles uppercase characters. The program is run against 6 tests or transactions producing the coverage matrix from t_1 to t_6 with test or transaction outcomes (error vector) for each of the transactions. Utilizing this information, the Ochiai SC for each statement is computed and the results are seen under the SC column in the same Figure. Based on the SC outcomes, the components that are potentially faulty have a higher SC value, and specifically in this example c_3 where the fault is located has the highest score.

$\mathcal{C}$	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	SC
	`function count(char *s) {`							
	`int let, dig, other, i;`	0	0	0	0	0	0	0
c_0	`let = dig = other = i = 0;`	1	1	1	1	1	1	0.87
c_1	`while (c = s[i++]) {`	1	1	1	1	1	1	0.87
c_2	`if('A'<=c && 'Z'>=c)`	1	1	1	1	0	1	0.93
c_3	**`let += 2;`**	1	1	1	1	0	0	1.0
c_4	`else if('a'<=c && 'z'>=c)`	1	1	1	1	0	1	0.93
c_5	`let += 1;`	1	1	0	0	0	0	0.71
c_6	`else if('0'<=c && '9'>=c)`	1	1	1	1	0	1	0.93
c_7	`dig += 1;`	0	1	0	1	0	0	0.71
c_8	`else if(isprint(c))`	1	0	1	0	0	1	0.47
c_9	`other += 1;}`	1	0	1	0	0	1	0.47
c_{10}	`printf("%d %d %d\n",`	1	1	1	1	1	1	0.87
	`let, dig, other);}`							
	Test case outcomes	1	1	1	1	0	0	

Figure 2.3: Example C Program, Activity Matrix and Ochiai diagnosis [2].

2.3 Feature Selection based Relief Approach (Relief)

The approach in this type of fault localization technique employs the feature selection algorithm Relief [17] in combination with a ranking algorithm [6] to compute the diagnosis results. Feature selection is a fundamental preprocessing step in machine learning which focuses on reducing the original feature datasets into smaller subsets with features that are effective and adequate to describe the concept at hand [18]. Data used in machine language is typically in a high dimensional feature space, described in the order of hundreds or thousands of features. A key issue with such data is that it increases the computation complexity of machine learning algorithms [19]. Therefore, feature selection operates by reducing the high-dimension-feature-space datasets to ones with decreased dimensionality while preserving vital features and possibly discarding irrelevant and redundant ones. In order to determine which features to retain or discard, an efficient method of evaluating feature relevance to the target concept is required. There are three approaches for evaluating the quality of candidate features, either through *filters*, *wrappers* or *embedded* approaches [6]. The filters approach discards redundant and irrelevant features, whereas the wrapper method selects a subset of features using learning algorithms. The wrappers often perform better than filters because the target learning algorithms are used in the feature selection process. Nonetheless, wrappers become time consuming as the dataset dimension feature space increases. In the embedded approach the feature selection occurs in parallel with the learning processes, typically in an iterative fashion and a good example of this category are the decision trees.

Among the existing feature weighting algorithms, the Relief algorithm is considered one of the most successful algorithms for evaluating the quality of features due to its simplicity and effectiveness. The key idea of Relief is to iteratively estimate feature weights according

to their ability to discriminate between neighboring patterns [20].

2.3.1 Relief Algorithm

Algorithm 1 as stated in the article by Roychowdhury and Khurshid [6] depicts an adaptation of the Relief algorithm proposed by Kira and Randell [18]. Relief uses the k-nearest-neighbors (KNN) classification principle [21], whereby samples or instances with identical classes are assumed to have similar relevant features in comparison to instances from different classes. In the algorithm, we assume that there are N instances or transactions in a P-dimensional feature space which means each instance X_k has P features (components) in its dataset, denoted by $X_k = \{x_{k1}, ..., x_{kP}\}$ where $k \in \{1, ..., N\}$. The algorithm works with data containing instances categorized over two classes. For each instance, Relief searches for the two nearest neighbor instances namely the *nearest hit* X^{h} and the *nearest miss* X^{m}. The nearest hit instance, X^{h}, belongs to a class that is identical to the instance X_k. While the nearest miss instance, X^{m}, is from a class that is different from that of the instance X_k. The neighbor instances are found using the Euclidean distance [18]. Thus, each instance is associated with a class category or label C_i, with $C_i \in \{1, 0\}$ (denotes failed or passed transactions). Using the input data for the algorithm, the relevance weight, W_i, for each individual feature (component) i is computed using the formula:

$$W_i = \frac{1}{N} \sum_{k=1}^{N} \frac{\delta(i,\, X_k,\, X^m)}{N} - \frac{\delta(i,\, X_k,\, X^h)}{N}, \tag{2.9}$$

where $\delta(i,\, X_k,\, X^h) = (x_{ki} - x_i{}^h)^2$ and $\delta(i,\, X_k,\, X^m) = (x_{ki} - x_i{}^m)^2$.

Algorithm 1: Relief Algorithm

```
Data: X: N samples or instances of P-dimensional data set
Data: C: Class Labels for each sample
Result: W: Returns a list of relevance values for the features
begin
    W ⟵ (0, ..., 0)
    for k ⟵ 1 to N do
        begin
            Randomly select a sample X_k
            Find nearest hit X^h, and nearest miss X^m
            for j ⟵ 1 to P do
                W[j] ⟵ W[j] − δ(j, X_k, X^h)/N + δ(j, X_k, X^m)/N
            end
        end
    end
end
```

2.3.2 Ranking Algorithm

The pseudo-code in Algorithm 2 illustrates how the ranking of components is obtained using the Relief method. The program under investigation has P components and N transactions as aggregates of the activity matrix X. The activity matrix is collect as in SFL and is then transposed such that the rows of X become transactions, while its columns become the components. Thus, $X[i, j] = 0$, implies that during the i^{th} transaction, the j^{th} component was not covered. The size of X is represented by the number of transactions and the number of components, $N \times P$. The outcome of the transactions is held in a list R. Each transaction is represented by either a pass (0) or a fail (1) in the class labels or error vector. The ranking process commences as follows:

1. Computing the relevance weights (list W) for each component using Relief.
2. Normalizing the relevance weights to lie in the set $[0.0..1.0]$ (list Z).
3. Translating the normalized weights using a floor function into intermediate ranks between 1 and P (list Q).
4. Finally, computing the score (final ranking) for each component i using list Q as follows:
$$score_i = \left(1 - \frac{Q[i]}{P}\right) \times 100,$$
For situations whereby multiple components have the highest score the score is computed using:
$$score_i = \left(\left(1 - \frac{Q[i]}{P}\right) \times CW\right) \times 100,$$
where
$$CW = \frac{P + 1 - |Q^{\#1}|}{P},$$
with $Q^{\#1}$ denoting the cardinality of the highest ranked components.

2.3.3 Example of Relief Diagnosis

In this section we show how Relief diagnosis is performed in practice. Table 2.3 is the activity matrix used earlier in the section about SFL as depicted in Figure 2.3, by transposing this activity matrix we get a new activity matrix with transactions as rows and components as columns in Table 2.4. The table comprises the input date for Relief, 6 transactions each denoted by $t_i = \{i \mid 1 \leq i \leq 6\}$, 11 components each denoted by $c_j = \{j \mid 0 \leq j \leq 10\}$ and an error vector or the transaction outcomes E. We feed the the transposed coverage matrix and error vector to the Relief method to produce the results in a list D. The row D in Table 2.4 holds the diagnosis results and the component $C2$ is ranked as highest and is, therefore, identified as the faulty component by Relief. Note that in Figure 2.3 Ochiai ranked $C3$ correctly as faulty. In this example, we can see that Ochiai has a better accuracy than Relief.

Algorithm 2: Ranking Algorithm

Data: X: Activity Matrix
Data: R: Result list
Result: Q: Returns a list of ranks
begin
  // Use a feature selection method: RELIEF and compute relevance values W_i
  for $i \longleftarrow 1$ **to** N **do**

$$Y[i] \longleftarrow W_i$$

  end
  // Normalize Y to capture the ranks; the line number with highest relevance value will be ranked 1.
  for $i \longleftarrow 1$ **to** N **do**

$$Z[i] \longleftarrow \begin{cases} \frac{Y[i]}{max(Y)} & \text{if } Y[i] \geq 0, \\ 0 & \text{if } Y[i] < 0, \end{cases}$$

  end
  // At this point values a $Z \in [0..1]$. Translate them to ranks using weighted interpolation. The ranks are stored in the list Q.
  for $i \longleftarrow 1$ **to** N **do**

$$Q[i] \longleftarrow \lfloor Z[i] + N \times (1 - Z[i]) \rfloor$$

  end
end

	t1	t2	t3	t4	t5	t6
c0	1	1	1	1	1	1
c1	1	1	1	1	1	1
c2	1	1	1	1	0	1
c3	1	1	1	1	0	0
c4	1	1	1	1	0	1
c5	1	1	0	0	0	0
c6	1	1	1	1	0	1
c7	0	1	0	1	0	0
c8	1	0	1	0	0	1
c9	1	0	1	0	0	1
c10	1	1	1	1	1	1
E	1	1	1	1	0	0

Table 2.3: Activity Matrix extracted from Figure 2.3.

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	E
t1	1	1	1	1	1	1	1	0	1	1	1	1
t2	1	1	1	1	1	1	1	1	0	0	1	1
t3	1	1	1	1	1	0	1	0	1	1	1	1
t4	1	1	1	1	1	0	1	1	0	0	1	1
t5	1	1	0	0	0	0	0	0	0	0	1	0
t6	1	1	1	0	1	0	1	0	1	1	1	0
D	0.0	0.0	0.248	0.0	0.0	0.0	0.0	0.124	0.0	0.0	0.0	

Table 2.4: Transposed Activity Matrix with Relief diagnosis results.

Chapter 3

SFL-Simulator

SFL-Simulator [14] is a tool developed using the programming language Ruby. Its main aim is to assist research in Spectrum-based Fault Localization. With the tool, a system component topology can be generated and its activations simulated to produce activity matrices[1]. An activity matrix can then be used as input for computing a diagnosis using the SFL technique of choice. The simulator already houses several SFL methods among others the similarity coefficients Ochiai, Jaccard and Tarantula. The simulator is extensible, for instance, for fault localization diagnosis methods that are not part of the tool, it can be used as a generator for the activity matrix input which can be fed into the new method to produce diagnosis results. Additionally, the system topology creation process can be performed using an input file containing the required system components. Another useful feature is the generation of pictures of a system topology as well as its activations.

3.1 Requirements

For the correct functioning of the SFL-Simulator tool, the machine under which the tool is run should have beforehand the following softwares installed:

- **Ruby version 1.9.3** [22]: A dynamic, open source programming language with an elegant syntax that is easy to write and read. Its main focus is to foster simplicity and productivity.
- **GraphViz** [23]: An open source graph visualization software.
- **Ruby-Graphviz gem** [24]: Using GraphViz, it provides an interface to layout and generate images of directed graphs in a variety of formats such as PostScript and PNG.
- **Colorize gem** [25]: An extension of the string class in Ruby that adds some methods to easily set color and text effect on console output.

[1] "Activity matrix" implies "Activity matrix with its error vector".

3.2 Structure

The tool consists of 15 classes and modules as depicted in Figure 3.1. Some of these are core classes and others are utility classes also known as modules in the Ruby language.

- **Component**: Represents a component or link in a system.
- **Trace**: Represents the traces of the activation of a component or a link.
- **TraceNode**: Represents a node (component or link) in a trace.
- **Topology**: Maintains components and links as well as their traces.
- **Activity**: Represents an activity matrix.
- **Actop**: Maintains topology activations.
- **Diagnosis**: Maintains the diagnosis results.
- **Similarity**: Contains utility operations for computing similarity coefficients.

The rest of the modules all serve as utility modules and their main function is to display the output of their corresponding classes and in some cases generate pictures of topologies.

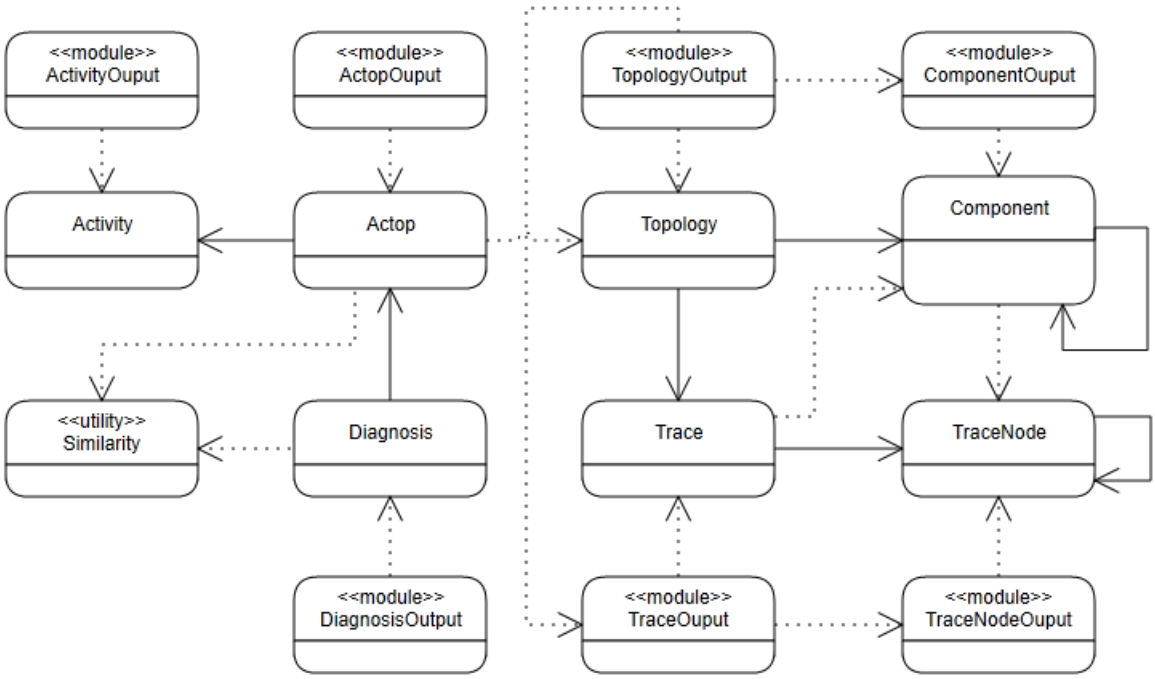

Figure 3.1: A high level class (modules) structure of the SFL-Simulator

3.3 Usage

3.3.1 System Topology Creation

A system topology comprises components and links. A new topology is created by instantiating the *Topology* class in the following fashion:

```
require './sfl_actop.rb'
require './sfl_diagnosis.rb'
t = Topology.new()
```

Following the creation of the system topology, components are added to it as shown in the ensuing snippet. Each component has a distinct name, health value and a failure probability. A component is added to the topology using the method *add(name, health value, failure probability)*. The health value is a value that denotes the probability of a component to produce a fault when it is activated during simulation and is in the range of [0.0..1.0]. A component that is absolutely faulty carries a health value of 0.0 while an absolutely non-faulty one has a value of 1.0. Any health value in between 0.0 and 1.0 represents the probability of the component being intermittently faulty, for example, a component with a 0.3 health value has a 70% chance of being intermittenly faulty at its activation. The failure probability [0.0..1.0] represents the probability on whether a fault is detected, leading to a failure. In a topology with components that have 0.0 failure probability values, a fault or faults are detected at the end of an activation. Addition of components to a topology takes the following form:

```
t.add("C0", 1.0, 1.0)
t.add("C1", 1.0, 1.0)
t.add("C2", 0.0, 0.0)
t.add("C3", 0.5, 1.0)
t.add("C4", 1.0, 1.0)
```

The components added to the topology interact with each other through links. Links function as connectors or communication channels connecting components with each other and similarly to components each link has a distinct name, as well as the adjacent components it connects and invocation probability which determines the likelihood of a link to be executed during a simulation. A link connects two components to each other or itself to a single component using the function *add(name, start component's name, destination component's name, link probability)*. The code snippet below shows how the components are linked in a topology.

```
t.link("L0", "C0", "C1", 1.0)
t.edge("L0", "C2")
t.link("L1", "C1", "C3", 0.8)
t.link("L2", "C2", "C3", 0.5)
t.link("L3", "C2", "C4", 0.3)
```

3.3.2 System Topology Picture

The next snippet shows how we can generate the pictorial representation of a system topology and Figure 3.2 depicts the picture of the system we have discussed in the preceding code snippets and section. Additionally, as shown in Figure 3.2, in the picture a system component is represented by a rectangular shape. The rectangular representation of the component contains four smaller compartments, the first is the name of the component e.g. *C0*, the second the health value *h=1.0*, the third is the failure probability *f=1.0*, and the last is for extra optional features and variables that can be assigned to the component. The link between the components is represented with a line with a circle in the middle. Inside the circle are the constructs, e.g. *L0=1.0* and *100* if the system is simulated, *L0* is the name of the link and *1.0* its link probability while the number *100* is the number of times the links is executed.

```
TopologyOutput.graph(t, :png, "ex_readme.png")
```

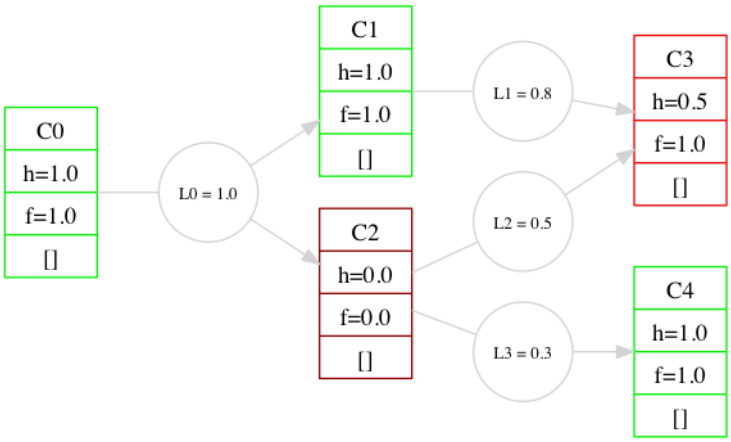

Figure 3.2: A system topology picture generated by SFL-Simulator.

3.3.3 Simulation Types

The execution of the components of the system can be simulated using several types of simulations:

- Using fixed number of executions or transactions. In this snippet 20 transactions are simulated starting from a single component *C*0.

```
t.activate_often(["C0"], 20)
```

- Using several components designated as starting components which means many components will probably be activated simultaneously. The snippet shows two components *C*0 and *C*4 both which may be activated at the same time.

```
t.activate_often(["C0", "C4"]){}
```

- Running the simulation until a component fails which is useful for simulating cases when the faulty component has very low fault intermittency.

```
t.activate_until_error(["C0"]){}
```

3.3.4 Traces

When components are activated in a system topology traces are produced and these traces can be displayed using the call:

```
TopologyOutput.traces(t)
```

This results in an output like in the next snippet. Each activation results in one line with the component executed, the link invoked, plus [fault, error, failure] information. The curly brackets indicate invocation nesting as the next snippet illustrates.

```
C0[0,0,0]{L0[0,0,0]{C1[0,0,0]}{C2[1,1,0]{L2[0,1,0]{C3[0,1,1]}}}}[fail]
C0[0,0,0]{L0[0,0,0]{C1[0,0,0]{L1[0,0,0]{C3[1,1,1]}}}{C2[1,1,0]}}[fail]
C0[0,0,0]{L0[0,0,0]{C1[0,0,0]{L1[0,0,0]{C3[0,0,0]}}}{C2[1,1,0]}}[fail]
C0[0,0,0]{L0[0,0,0]{C1[0,0,0]}{C2[1,1,0]}}[fail]
C0[0,0,0]{L0[0,0,0]{C1[0,0,0]}{C2[1,1,0]{L2[0,1,0]{C3[0,1,1]}}}}[fail]
C0[0,0,0]{L0[0,0,0]{C1[0,0,0]{L1[0,0,0]{C3[0,0,0]}}}{C2[1,1,0]}}[fail]
```

3.3.5 System Topology Activations

An activated system topology can be utilized for diagnosis experiments. First, the system topology activation is created and we can generate a picture to show the activations as in Figure 3.3 and the activity matrix as shown in the second snippet:

```
actop = Actop.new(t)
ActopOutput.screen(actop)
ActopOutput.graph(actop, :png, "ex_readme_actop.png")
```

```
C0 11111111111111111111
C1 11111111111111111111
C2 11111111111111111111
C3 01111110110010111111
C4 10100010001011101111
L0 11111111111111111111
L1 01111110010000111111
L2 01011000100010000111
L3 10100010001011101111
 E 11111111111111111111
```

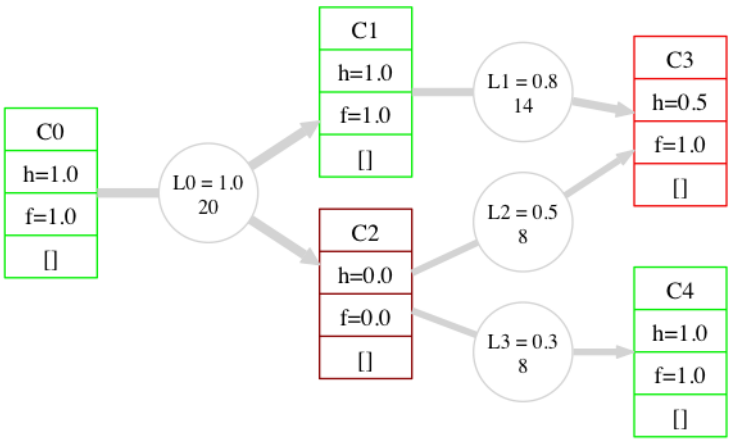

Figure 3.3: A system topology activation picture generated by SFL-Simulator

3.3.6 Diagnosis Results

Using the system topology activation data, diagnosis results can be computed (first snippet) and produce output (second snippet) as follows:

```
diagnosis = Diagnosis.new(actop)
DiagnosisOutput.screen(diagnosis, {:sort => :ochiai}, :ochiai, :jaccard)
```

Underneath are the diagnosis results of two similarity coefficient based methods namely Ochiai and Jaccard.

```
   |:ochiai|:jaccard|
C0 | 1.000 | 1.000 |
C1 | 1.000 | 1.000 |
C2 | 1.000 | 1.000 |
L0 | 1.000 | 1.000 |
C3 | 0.975 | 0.950 |
L1 | 0.806 | 0.650 |
L2 | 0.707 | 0.500 |
C4 | 0.387 | 0.150 |
L3 | 0.387 | 0.150 |
```

3.3.7 Contribution

As part of the work performed in this study, our contribution to the tool is the implementation (in Ruby language) of the Feature Selection based fault localization method Relief, see Appendix A. In the implementation, we use the same input data as in SFL. However, the activity matrix has to be transposed before it is fed into Relief. So using the same coverage matrix used in Section 3.3.5 but transposed, we can compute the diagnosis results for Relief as follows:

```
matrix = transpose of activity matrix
error_vector = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
components_array = ["C0", "C1", "C2", "C3", "C4", "L0", "L1", "L2", "L3"]
scores = compute_relief_scores(matrix, error_vector, components_array)
```

The resulting diagnosis table is:

```
C0 | 0.0
C1 | 0.0
C2 | 0.0
C3 | 0.0
C4 | 0.0
L0 | 0.0
L1 | 0.0
L2 | 0.0
```

The Relief diagnosis results depicted in this example show no suspicion information about the possible faulty components. This can be attributed to the way the ranking algorithm is designed. It does not consider negative weights, all components with negative relevance weights receive zero scores. This happens mostly when the outcomes of the tests or transactions are all of the same class either all pass or fail and in this particular example the outcomes were all fail.

Chapter 4

Experimental Setup

In this chapter we describe the setup of our experiments and define how the fault localization diagnostic performance or effectiveness is measured. The chapter is organized as follows. First, Section 4.1 presents the systems we use in the experiments. Then Section 4.2 lists the parameters that can have influence on the results of the experiments followed by how the parameters interact with each other in Section 4.3. In Section 4.4 we explain the experimental design. Finally, Section 4.5 discusses which performance metric we use to measure the diagnosis accuracy of the fault localization methods to be compared.

4.1 Systems

In our experiments we simulate 8 different basic systems. The systems contain disparate number of components connected with each other through links in various ways. The 8 basic systems produce roughly about 512 simulated systems. Table 4.1 contains details about the number of components and links per a basic system used and Figure 4.1 illustrates a pictorial representation of the topology of one of the 8 systems, for rest of the systems check Appendix B.

Topology	Number of components	Number of links
1	6	7
2	6	11
3	8	7
4	14	18
5	8	16
6	16	27
7	8	11
8	8	21

Table 4.1: Systems and their constituent components and links used in the experiments.

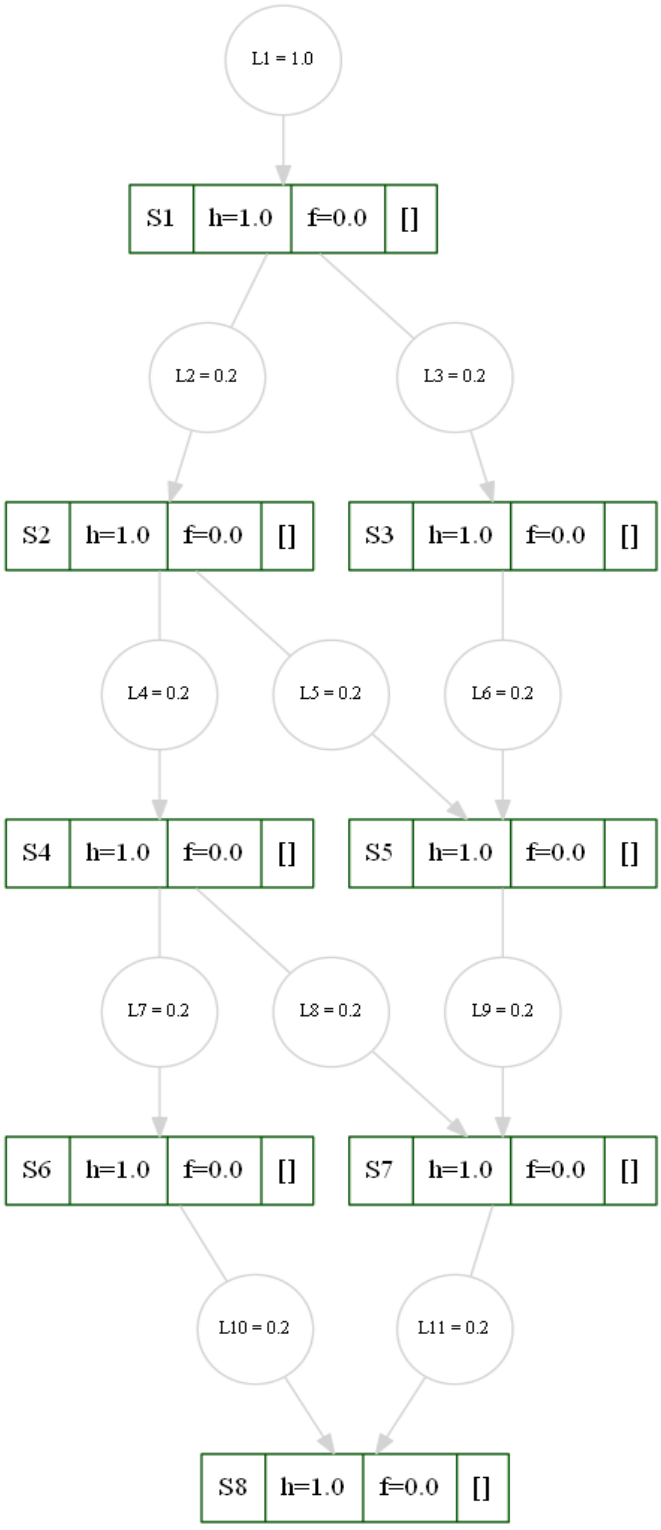

Figure 4.1: System 7 with 8 components and 11 links.

4.2 Parameters

The experiments involve many parameters and an uncontrolled interaction between two or more of these parameters may have undesirable effects on the experimental results. Therefore, it is of paramount importance to identify those parameters that influence the diagnostic accuracy of the methods to be compared during the experimental design stadium. The following subsections describe the parameters that we consider in our experimental procedures.

4.2.1 Faults

Faults describe which components are set as faulty in a system using their health values. The number of faults is directly equivalent to the number of faulty components in a system. In our experiments, each system is simulated with a single fault, two faults and three faults. Additionally, two randomly chosen systems are executed with all components designated as faulty.

4.2.2 Health Value (h)

The health value (h) denotes the probability of a component to be faulty during its activation. The experiments use 7 different health values $h = \{0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 0.9\}$ per a faulty component. In case of multiple faults, the faulty components are assigned identical health values. For example, if we have an experiment involving 2 faulty components and the heath value 0.6, each of the faulty components has its health value set as $h = 0.6$. For all non-faulty components $h = 1.0$ throughout all experiments.

4.2.3 Link Probability (lp)

We use in the experiments 5 values, $\{0.2, 0.4, 0.5, 0.6, 0.8\}$, to designate the link probabilities (lp) between components. Like the health values, the links in a system are iteratively assigned the same probability value except for the starting link (from where execution starts). The starting link is connected to a single component as an incoming link and is always assigned a probability value 1.0.

4.2.4 Transactions

Each system is executed using 25, 100 and 200 transactions. However, in case of experiments where a system is run up to until the point where an error is detected, the number of transactions are simulation-dependent and are determined only after the simulation terminates.

4.3 Interaction between parameters

The parameters identified in the previous sections certainly interfere with each other during the experiments. It is therefore important to vary the value of only one of these parameters

while keeping the rest unaltered during any given experiment. In our experiments, the emphasis is on the effects of the number of faults, health probability and link probability on the experimental results. Thus, these three parameters are our primary factors, whereas the transactions are secondary.

4.4 Experimental Design

In Figure 4.2 we depict the design of the experimental procedure in our study using the parameters discussed in the previous section. The steps are as follows:

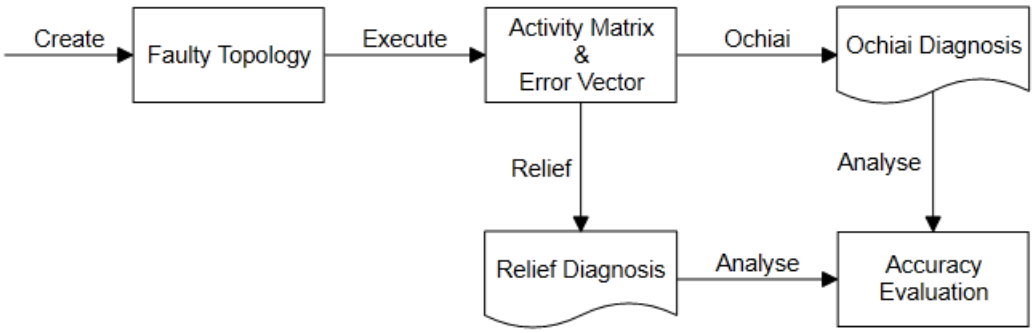

Figure 4.2: Experimental design steps.

1. **System Topology Creation**
   We create for each of the 8 basic systems several system instances using the corresponding values for health and link probabilities as well as the number of faults, as shown in Table 4.2. Each system instance or topology is created using one, two and three faults respectively by assigning the faulty components identical health values. In 2 randomly chosen systems we set all the components as faulty. For each of the three transactions types, 25, 100 and 200, each system instance is executed 500 times resulting in 500 individual diagnoses. In addition, the instances from 7 systems are simulated with a single fault using executions that terminate as soon as an error is detected.

2. **Acquisition of Activity Matrices and Error Vectors** The activity or coverage matrix produced from a system instance execution consists of spectra obtained using the block hit program spectra method. The components are represented by binary values, 1 if a component has been activated during an execution and 0 otherwise. The system instances each produce 500 activity matrices for each of the 25, 100 and 200 number of transactions. The number of transactions is excluded in the simulations that run until a single error is detected because their transactions are simulation-dependent. Each matrix has its corresponding error vector which holds the outcomes of the transactions.

Number of faults	Health Values(h)	Link Probability(lp)	Transactions
1	0.0	0.2	25
2	0.2	0.4	100
3	0.4	0.5	200
	0.5	0.6	
	0.6	0.8	
	0.8		
	0.9		

Table 4.2: System topology creation variables.

3. **Acquisition of Diagnosis Results** The generated activity matrices and their corresponding error vectors are used as input and fed to Ochiai to yield diagnosis results. Furthermore, we use the same matrices but in the transposed position together with the error vectors to produce Relief diagnosis results.

4. **Diagnosis Results Analysis** Both Ochiai and Relief rank components according to their likelihood of being faulty. Thus, the results obtained from both methods are analyzed by looking at the highly ranked components in correspondence to components that are presumably faulty in the system. The diagnosis results are classified into four categories: UNIQUE, INCORRECT, AMBIGUOUS and UNDIAGNOSED. A UNIQUE diagnosis represents a result in which the faulty components are correctly identified. An INCORRECT diagnosis characterizes a diagnosis result in which the faulty components are partial or wholly not detected. An AMBIGUOUS diagnosis is one in which more components are identified as faulty than there are faulty components. To explain these concepts, we use Tables 4.3 and 4.4 which depict diagnosis results of Ochiai and Relief side by side. The diagnosis results are from a system in which three components namely S_2, S_3 and S_6 are faulty. The results list the components in their likelihood of being faulty in descending order. Since the system has three faulty components, we are interested in the top three highest ranked components. Therefore, we inspect whether the faulty components are actually ranked as the three highest components in the diagnosis results. In case of Table 4.3, Relief identifies the three components correctly while Ochiai only identifies two faulty components out of three. Thus, we categorize the Relief diagnosis as UNIQUE and Ochiai as INCORRECT. In Table 4.4 Ochiai diagnosis is this time UNIQUE, whereas Relief has four components ranked as highest ($S3$ and $S4$ share same ranking) which equates to an AMBIGUOUS diagnosis as $S4$ is not one of the three faulty components. In situations where experiments produce an error vector with all transactions outcomes as passed, we categorize the diagnosis as UNDIAGNOSED.

Ochiai		*Relief*	
S2	0.725	S2	0.781
S1	0.7	S6	0.521
S3	0.674	S3	0.26
S5	0.621	S4	0.195
S4	0.583	S5	0.065
S6	0.523	S1	0.0

Table 4.3: INCORRECT Ochiai and UNIQUE Relief diagnosis in a system with 3 faulty components.

Ochiai		*Relief*	
S2	0.725	S2	0.781
S6	0.7	S6	0.521
S3	0.674	S3	0.26
S5	0.621	S4	0.26
S4	0.583	S5	0.065
S1	0.523	S1	0.0

Table 4.4: UNIQUE Ochiai and AMBIGUOUS Relief diagnosis in a system with 3 faulty components.

4.5 Performance Metric (PM)

The performance metric (PM), also known as the quality metric, measures the diagnosis accuracy or effectiveness of a fault localization technique (in this study Ochiai or Relief) to uniquely identify faulty components in a system. We measure the performance metric as the percentage of the number of UNIQUE diagnoses against the total number of diagnoses simulations, 500 in most of our experiments. A diagnosis that falls under the UNDIAGNOSED category is excluded from the computation. In equation 4.1 *unique* represents the number of correct diagnoses, *diagnoses* is the number of simulations (activity matrices) which is 500 in most of our experiments and *undiagnosed* is the number of diagnoses that are designated as UNDIAGNOSED.

$$Performance\ Metric(PM) = \frac{unique * 100}{diagnoses - undiagnosed} \tag{4.1}$$

Chapter 5

Experimental Results

In the ensuing sections of this chapter, we present the findings from the experiments performed using the SFL-Simulator. This chapter commences with the experimental results from systems with single faults in Section 5.1, followed by Section 5.2 which deals with systems comprising two faults. Section 5.3 presents results from systems with three faults. Finally, experiments where all the components in a system are set as faulty is discussed in Section 5.4.

5.1 Systems with one fault

Two different simulation approaches have been used in experiments for single faults. This section explores these two different sets of experiments.

5.1.1 Experiments with predetermined number of transactions

The experiments were performed using 8 systems and each system was executed using 7 health probability values (h), 5 link probability values (lp) and 3 types of transactions with 25, 100 and 200 number of transactions respectively. The aforementioned values created 840 combinations or system instances[1] which were simulated 500 times each. Thus, the total number of simulations[2] realized in these experiments were 420,000. All the plots presented in this chapter express the results obtained using a specific health value and a number of transactions. Furthermore, the x-axis embodies the link probability values and the y-axis represents the percentage of the correct diagnoses.

As can be seen from the graph, Figure 5.1, regardless of the link probability and the number of transactions used, at $h = 0.0$ both Ochiai and Relief exhibit the same diagnosis effectiveness.

25 transactions and $h > 0.0$: Results from the majority of the systems show that Relief is either better than or comparable to Ochiai at $lp \leq 0.3$. Relief performance diminishes as

[1] A system instance is a system configuration with a specific number of faults with a specific health probability value and links with a specific link probability value.

[2] Simulation is used here to also mean diagnosis result

lp increases and at $lp > 0.4$ Ochiai performance starts to improve and around $lp \geq 0.5$ it starts to significantly outperform Relief as depicted in Figure 5.2. In a few cases the results illustrate a comparable Ochiai and Relief performance at higher h or lp values or both. Only one system shows results where Ochiai is better than Relief at $h = 0.2$ under almost all lp values. In the same system at $h > 0.2$, Relief shows better diagnosis performance than Ochiai, however, in a few cases at a higher link probability ($lp \geq 0.8$) and health probability values ($0.8 \leq h \leq 0.9$), Ochiai does better than Relief and an example of that is shown in Figure 5.3.

100 transactions and $h > 0.0$: When the systems are executed with 100 transactions and $h > 0.0$, as Figure 5.4 depicts, in most cases at lower link probabilities, Ochiai starts with a poorer performance than Relief. At $0.2 \leq lp \leq 0.5$ Ochiai gradually improves and from $lp \geq 0.5$ it shows relatively better diagnosis effectiveness over Relief. Furthermore, in most cases with $0.8 \leq h \leq 0.9$ Ochiai is more effective compared to Relief. Similarly as in the case of 25 transactions, one system as shown in Figure 5.5 presents Relief as superior to Ochiai with $h > 0.2$ under all link probabilities.

200 transactions and $h > 0.0$: Like in the case of 100 transactions, results from one system always show that Relief outperforms Ochiai from $h > 0.2$. Also in most cases the two methods are comparable or Ochiai does worse than Relief at lower lp values, but Ochiai starts showing a better diagnosis effectiveness than Relief at higher link probabilities from around $lp \geq 0.5$, and this can be seen in Figure 5.6 which shows Ochiai having better performance from $lp \geq 0.6$.

5.1.2 Experiments with single error detection

Experiments under this simulation type differ from the one with planned number of transactions only in the fashion in which the number of transactions is determined. The three types of transactions are excluded and the simulations are run until an error is detected. Specifically, this means that the error vector produced from each simulation comprises at most a single failed outcome while the rest are pass. The total number of simulations was lower compared to its planned transactions counterpart, 140,000 simulations from 7 systems.

As illustrated in Figures 5.7 and 5.8, the results from all the involved systems show that Ochiai consistently outperforms Relief. In a very few cases, Relief showed a slightly[3] better performance than Ochiai at low link probability with $lp \leq 0.2$.

5.2 Systems with two faults

Experiments using 8 systems with two faults yielded 420,000 simulations. For 25, 100 and 200 transactions at $h = 0.0$, 7 out of 8 systems yield results where Relief considerably outperforms Ochiai as displayed in Figure 5.9. In only 1 system we have seen results that show a comparable performance between the two methods and in another Ochiai performs better than Relief at $lp < 0.4$.

[3] In our results interpretation "slightly" is used to denote a difference of $\leq 1.0\%$ in diagnosis performance.

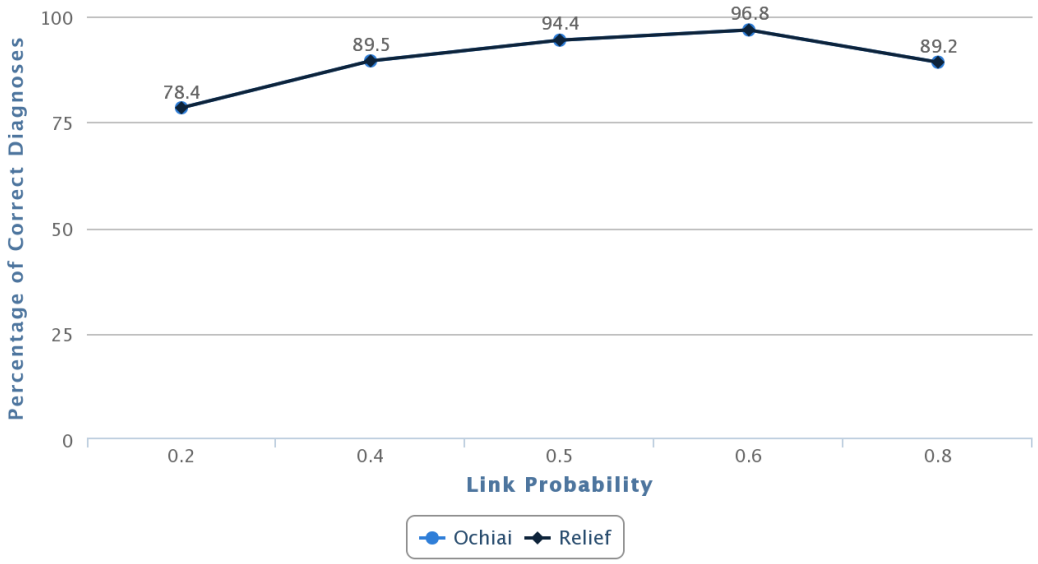

Figure 5.1: Comparison of 500 diagnosis results between Ochiai and Relief from System 3 with single fault ($h = 0.0$) and 25 transactions.

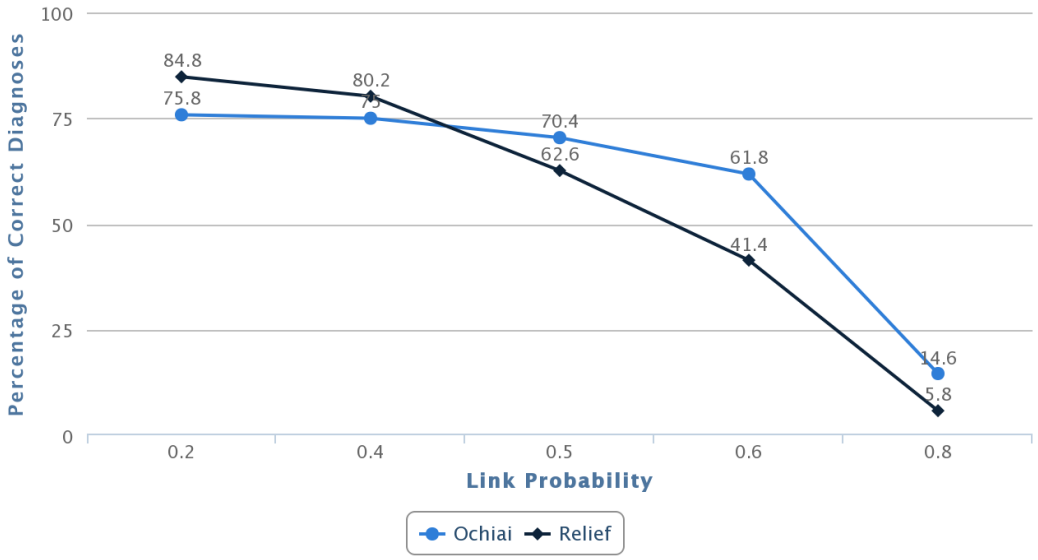

Figure 5.2: Comparison of 500 diagnosis results between Ochiai and Relief from System 6 with single fault ($h = 0.5$) and 25 transactions.

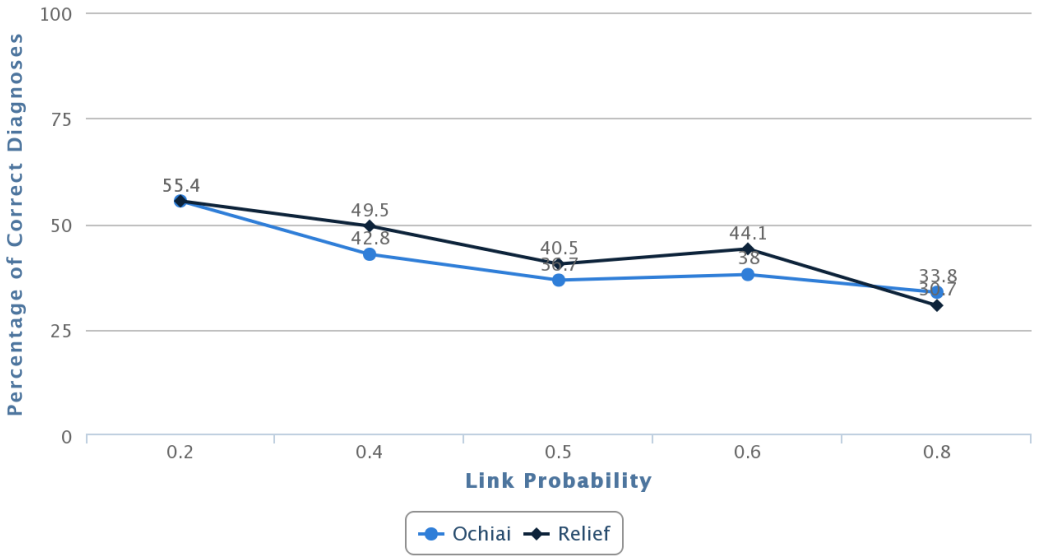

Figure 5.3: Comparison of 500 diagnosis results between Ochiai and Relief from System 3 with single fault ($h = 0.8$) and 25 transactions.

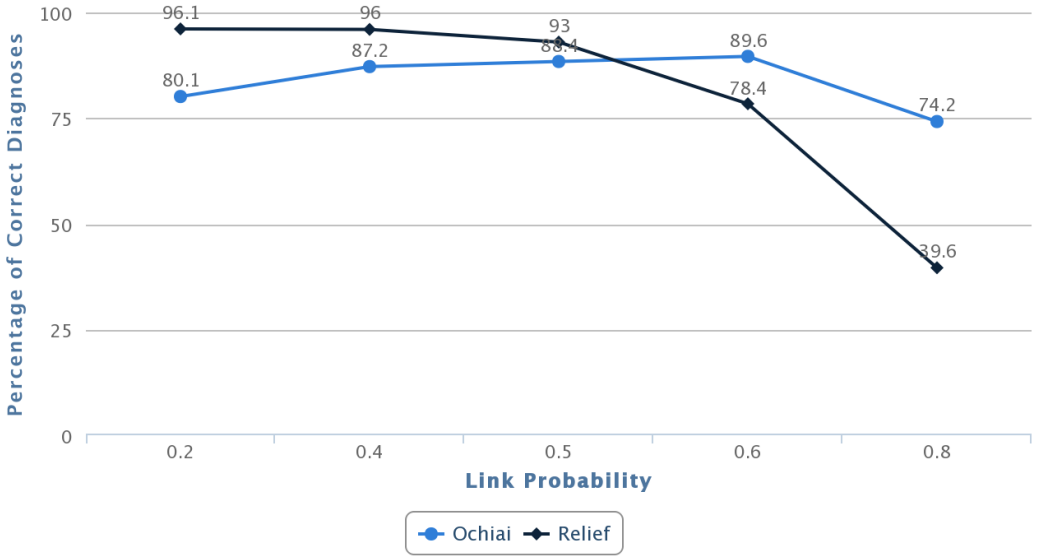

Figure 5.4: Comparison of 500 diagnosis results between Ochiai and Relief from System 4 with single fault ($h = 0.8$) and 100 transactions.

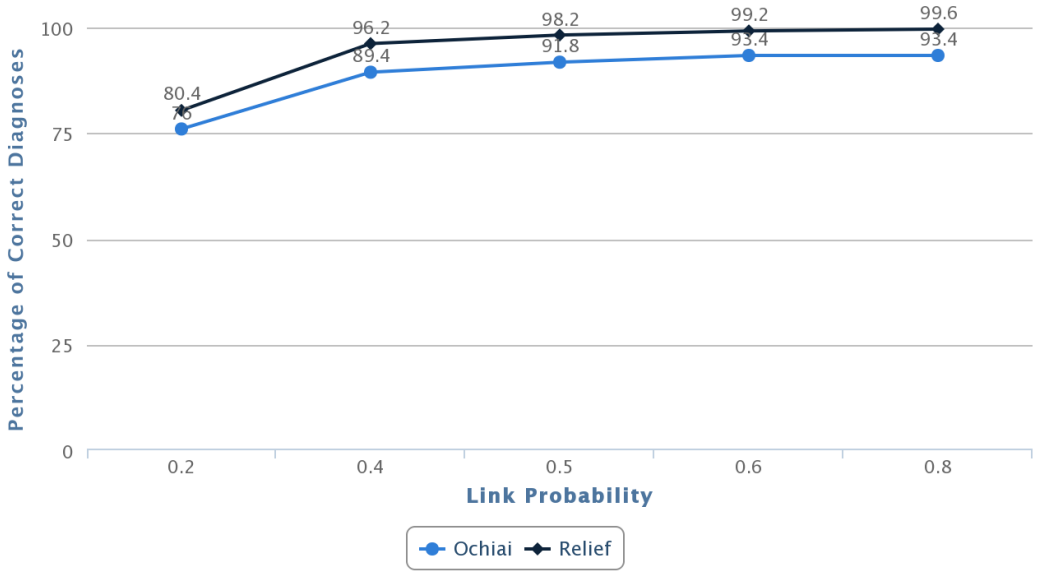

Figure 5.5: Comparison of 500 diagnosis results between Ochiai and Relief from System 3 with single fault ($h = 0.6$) and 100 transactions.

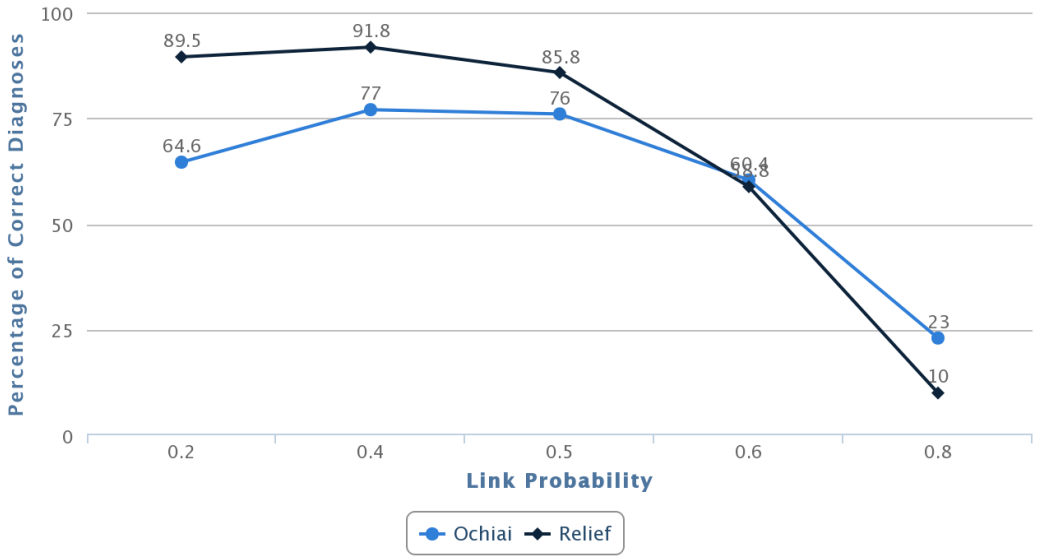

Figure 5.6: Comparison of 500 diagnosis results between Ochiai and Relief from System 6 with single fault ($h = 0.9$) and 200 transactions.

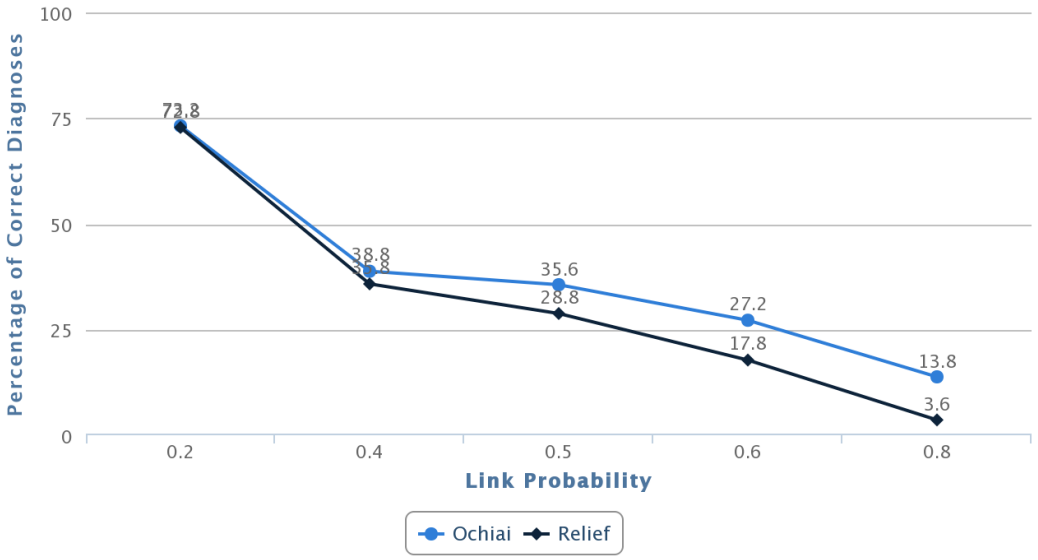

Figure 5.7: Comparison of diagnosis results between Ochiai and Relief from System 4 with single fault ($h = 0.5$) and simulation until error is detected.

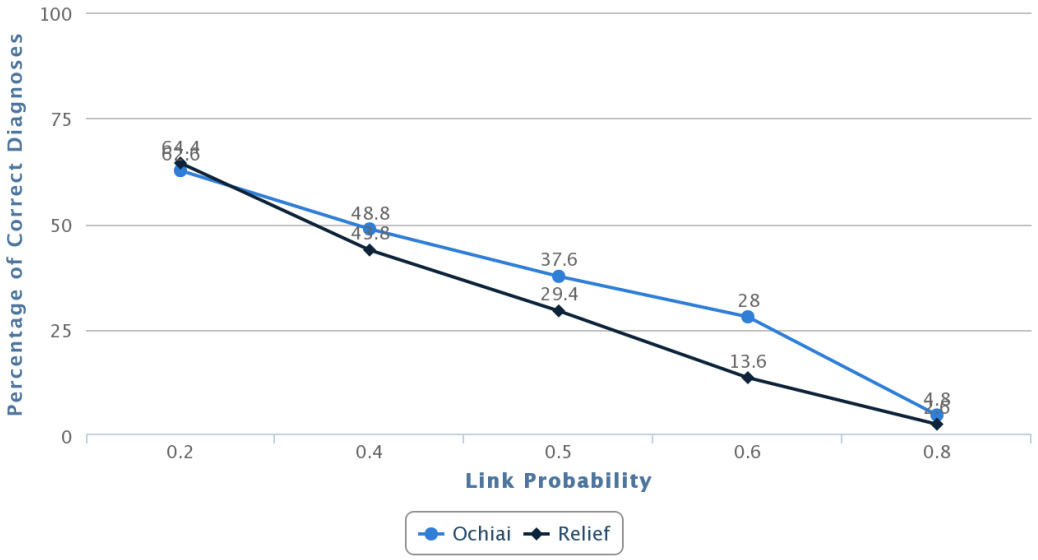

Figure 5.8: Comparison of diagnosis results between Ochiai and Relief from System 2 with single fault ($h = 0.9$) and simulation until error is detected.

25 transactions and $h > 0.0$: As shown in Figure 5.10, at lower link probabilities $0.2 \leq lp \leq 0.5$ and $0.2 \leq h \leq 0.4$ Ochiai is reasonably better than Relief. In a case or two Ochiai does slightly better until $lp < 0.6$ and then as lp further increases Relief improves over Ochiai. For $h > 0.4$, in more than half of the systems the results show that Ochiai outperforms its counterpart, while in the other cases Relief outperforms Ochiai. In a few cases the two methods perform equally.

100 transactions and $h > 0.0$: In the majority of the systems Ochiai consistently outperforms Relief at $h \leq 0.4$ and $lp \leq 0.2$. While at $lp > 0.2$ Relief yields better performance than Ochiai. However, in one system Ochiai was superior to Relief at $h = 0.4$ under all lp values. Generally, as h increases, $h \geq 0.5$, Ochiai performs better than Relief in the majority of the results, this is depicted in Figure 5.11. In a few cases Ochiai does better at lower lp while Relief at higher lp values.

200 transactions and $h > 0.0$: The results at $0.2 \leq h \leq 0.4$ are similar to those discussed in the preceding section about 100 transactions. In addition, one system shows alternating results between the two methods. Similarly to the results in 100 transactions at $h > 0.4$, Ochiai outperforms Relief in a majority of the cases with increase in h as can be seen in Figure 5.12. In a number of cases Ochiai performs well at lower lp values ($lp \leq 0.4$) then as the lp values increase it begins to do worse than Relief.

5.3 Systems with three faults

Similarly to experiments using two faults, experiments with 8 systems comprising three faults produced 420,000 simulations. For all three types of transactions namely 25, 100 and 200 the results show a reasonable consistent trend. Up to until $h \leq 0.4$ as in Figure 5.13, in majority of the results Relief outperforms Ochiai. In a few cases Ochiai shows better results than Relief at $lp \leq 0.2$ and in some cases the two perform equally. From $h \geq 0.5$ only a single system produces results that exhibit Ochiai as superior to Relief as demonstrated in Figure 5.14, and the rest of the results are dominated by the Relief method.

5.4 Systems with all components faulty

We performed experiments on only two randomly chosen systems that collectively produced 105,000 simulations where all components are designated as faulty. For all types of transactions namely 25, 100 and 200 and health values as well as all link probabilities, Ochiai outperforms Relief. Figures 5.15 and 5.16 depict a subset of these results where you can see how Relief undoubtedly performs worse than Ochiai.

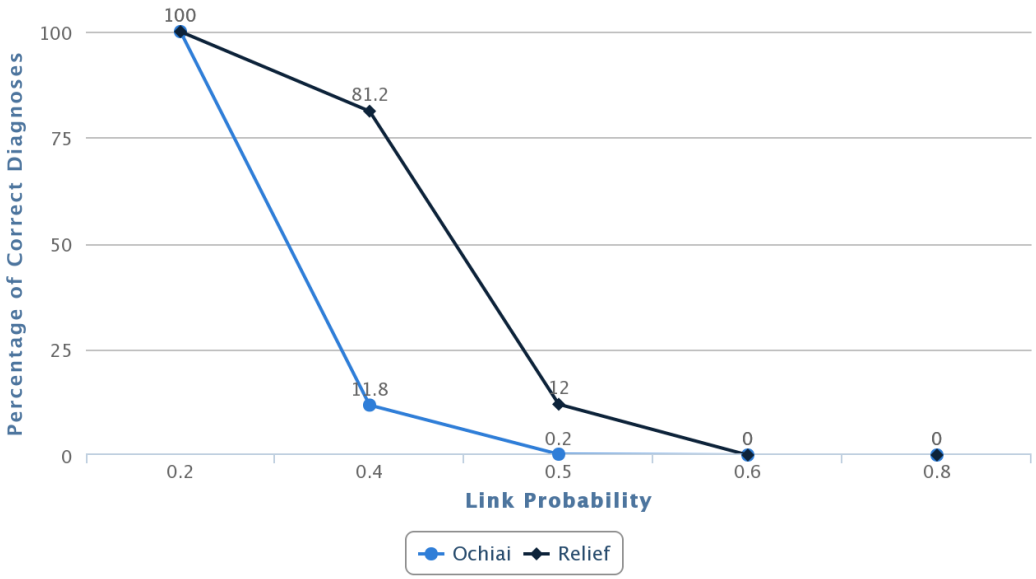

Figure 5.9: Comparison of 500 diagnosis results between Ochiai and Relief from System 2 with two faults ($h = 0.0$) and 200 transactions.

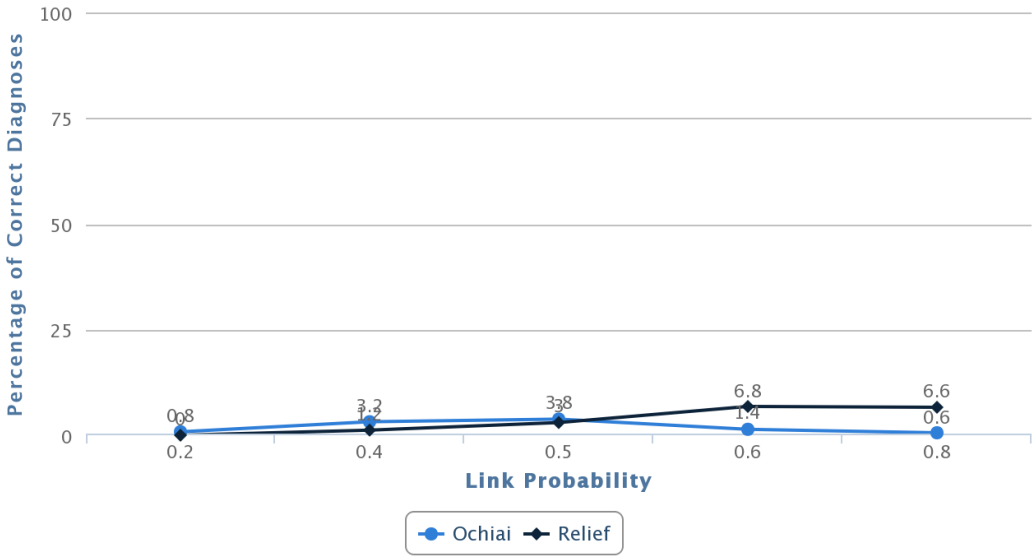

Figure 5.10: Comparison of 500 diagnosis results between Ochiai and Relief from System 7 with two faults ($h = 0.4$) and 25 transactions.

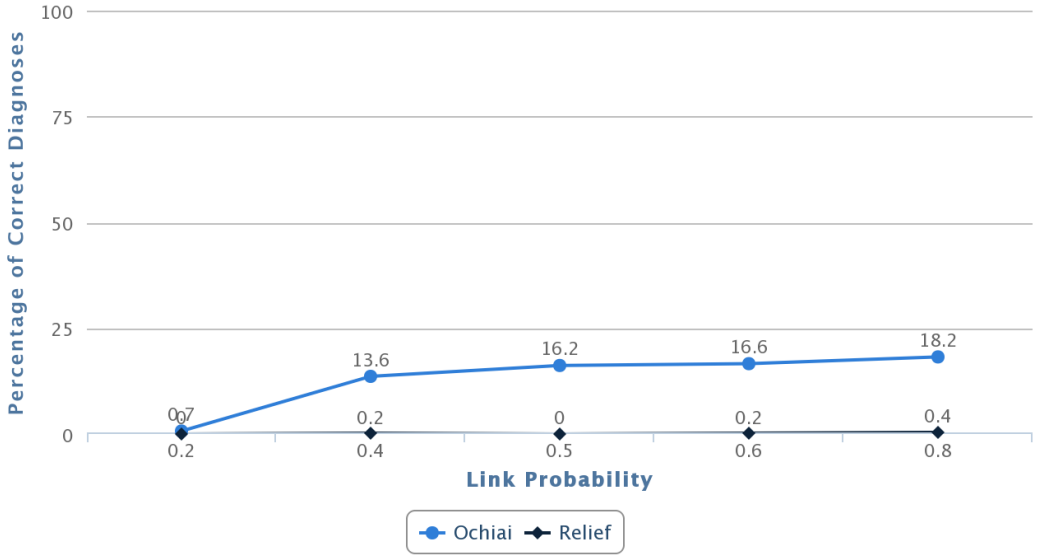

Figure 5.11: Comparison of 500 diagnosis results between Ochiai and Relief from System 3 with two faults ($h = 0.6$) and 100 transactions.

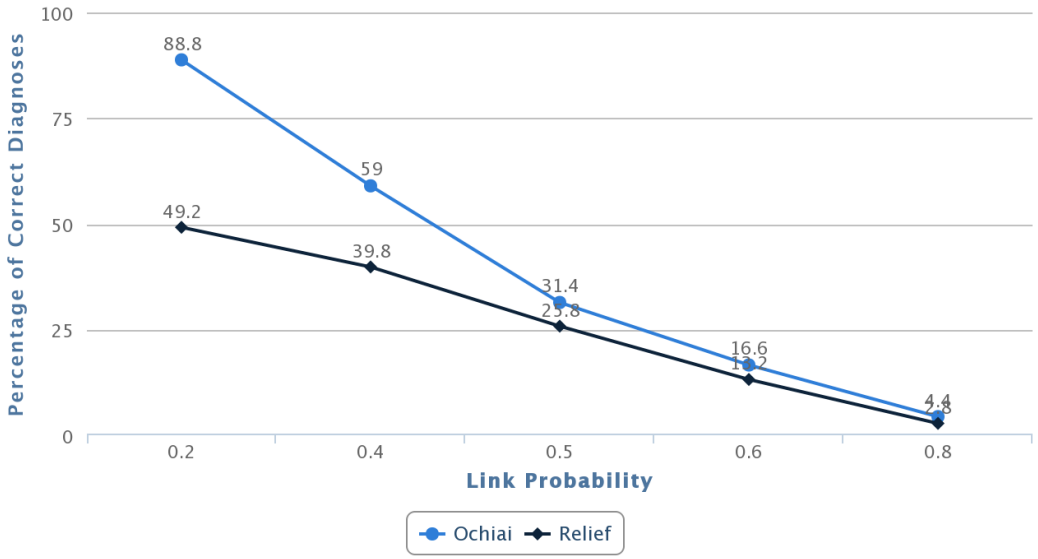

Figure 5.12: Comparison of 500 diagnosis results between Ochiai and Relief from System 6 with two faults ($h = 0.6$) and 200 transactions.

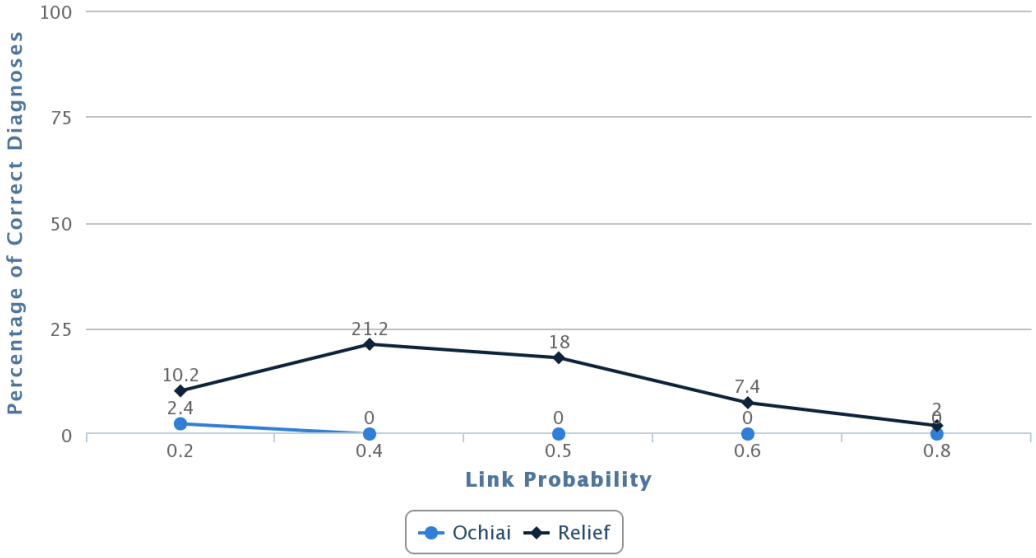

Figure 5.13: Comparison of 500 diagnosis results between Ochiai and Relief from System 5 with three faults ($h = 0.4$) and 100 transactions.

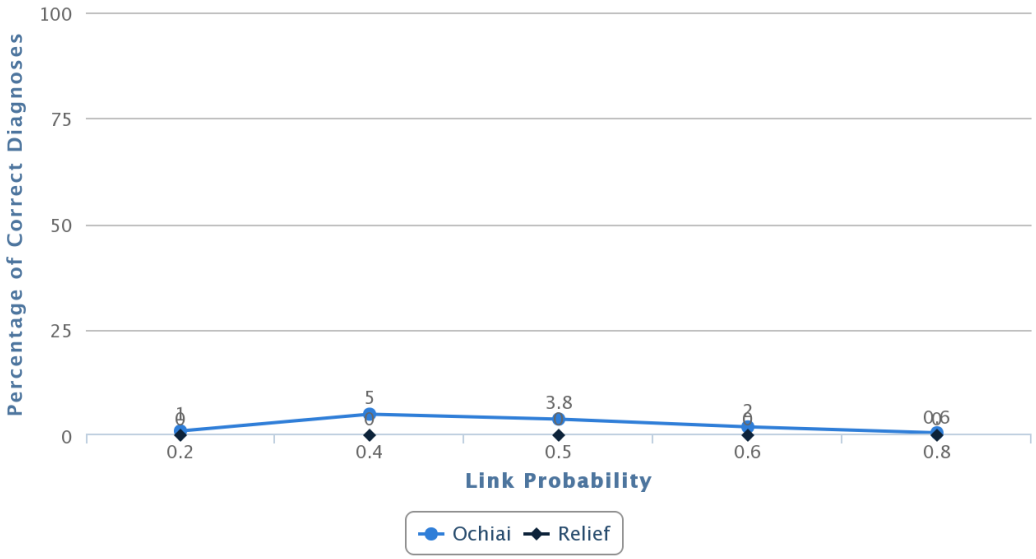

Figure 5.14: Comparison of 500 diagnosis results between Ochiai and Relief from System 3 with three faults ($h = 0.8$) and 200 transactions.

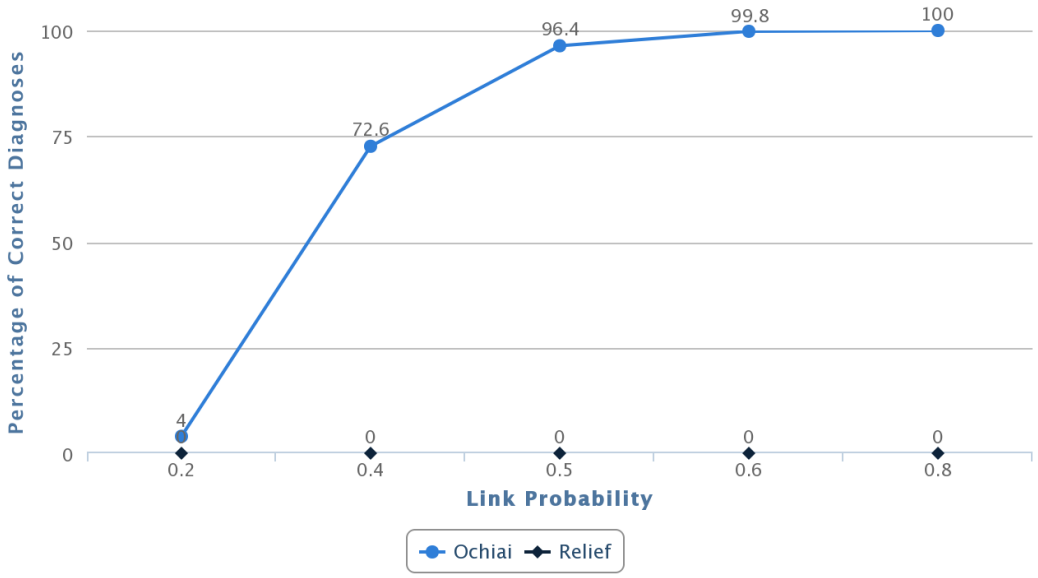

Figure 5.15: Comparison of 500 diagnosis results between Ochiai and Relief from System 7 with all components faulty ($h = 0.6$) and 25 transactions.

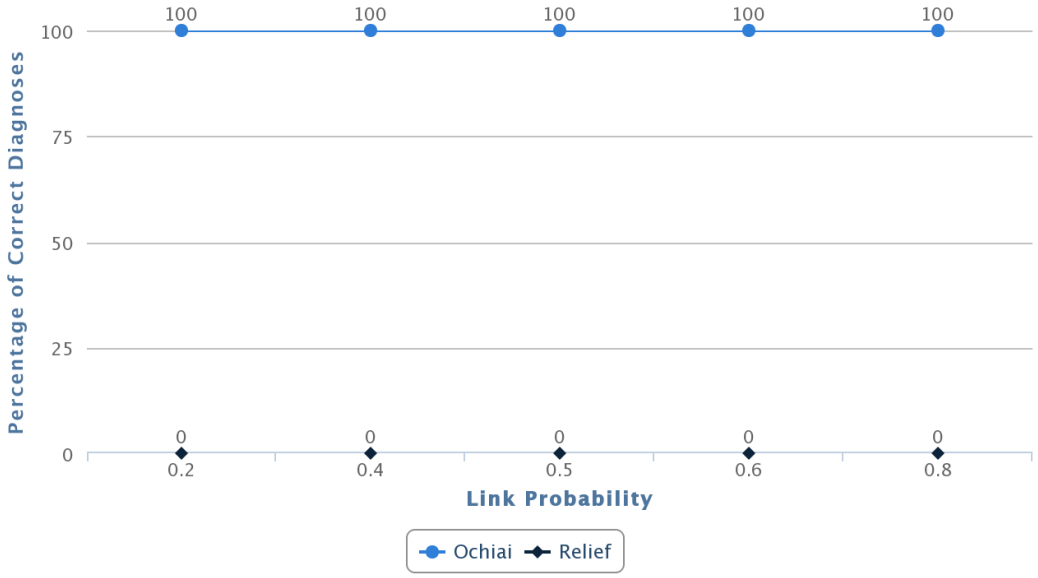

Figure 5.16: Comparison of 500 diagnosis results between Ochiai and Relief from System 8 with all components faulty ($h = 0.0$) and 200 transactions.

Chapter 6

Discussion of Results

In the previous chapter, we presented the results from our experiments. Our next task is to give an interpretation of these results. We start with single faults in Section 6.1 and then multiple faults in Section 6.2. Finally, in Section 6.3 we explain the time and space complexity of Ochiai and Relief.

6.1 Single faults

When the health probability of a faulty component is notably low, i.e. $h = 0.0$, this means that the component is 100% faulty. Under this condition, both Ochiai and Relief identify the faulty components correctly with the same effectiveness. At such a low h value any involvement of a faulty component in a transaction leads to an eminent failed transaction and as a result the spectrum of the faulty component largely resembles the corresponding error vector. Ochiai tends to deliver highly accurate diagnoses in such conditions and based on our observations, we can affirm that at $h = 0.0$ Relief behaves precisely like Ochiai. For Ochiai, this phenomenon has also been confirmed by Abreu et al. [4].

Observation 6.1.1. *Components with spectra that most resemble the error vector are most likely to be ranked the highest by Ochiai.*

Observation 6.1.2. *In systems with single faults without fault intermittency (h = 0.0), Relief acts precisely like Ochiai in Observation 6.1.1*

In cases where $h > 0.0$ combined with a few[1] transactions, we have seen fluctuating results. With low link probability values, $lp \leq 0.3$, Relief performs reasonably well compared to Ochiai. Due to low lp combined with higher h, many components are hardly involved in transactions and this leads to sparsity in activity matrices and error vectors. The implications for Ochiai is that non-faulty components with the least activations but largely involved in failed transactions, can mistakenly be ranked as highest as established in Observation 6.1.1, while concurrently exonerating faulty components that have many activations that lead to fewer failed transactions. Evidently, in Relief the highest ranked components

[1] Not more than 25 transactions.

are those that are largely activated in the nearest miss transactions and hardly in nearest hit transactions. This confirms the fact that the nearest miss transactions contribute to increase in the relevance weights of components because they denote the frequent involvement of the component in change of classes or involvement in failed transactions. As lp increases components become regularly activated yielding coverage matrices that are densely populated with 1s and the corresponding error vectors are more likely to comprise more 1s than in case of lower lp values. Under these conditions, we observed that Ochiai usually ranks the faulty components correctly. Intuitively, these components are often the ones that have the highest *LexicoRat* values ordered lexicographically. *LexicoRat* denotes a combination of the number of failed transactions that components are involved in and their *Rat* values, where for a component j

$$Rat(j) = \frac{Number\ of\ failed\ transactions\ involving\ j}{Total\ number\ of\ activations\ of\ j} \tag{6.1}$$

$$LexicoRat(j) = [Number\ of\ failed\ transactions\ involving\ j, Rat]. \tag{6.2}$$

The poor performance of Relief can be ascertain to the dense activity matrices and error vectors. Diagnosis results are more likely to produce ambiguous diagnoses or non-faulty components are frequently awarded higher relevance weights because many components will probably be involved in transactions that resemble each other.

Observation 6.1.3. *Low link probabilities are likely to produce sparsely populated activity matrices. While high link probabilities are likely to generate dense activity matrices.*

Observation 6.1.4. *At $h > 0.0$, fault intermittency is introduced and the activation of faulty components does not necessarily lead to failed transactions.*

Observation 6.1.5. *For Relief, the components that are ranked the highest are mostly activated in the nearest miss transactions and rarely in the nearest hit transactions.*

At $lp \geq 0.8$ and $0.8 \leq h \leq 0.9$ the interactions are producing densely populated activity matrices but due to high fault intermittency the error vector contains hardly any errors as mentioned in Observation 6.1.4. This increases the chance of Relief upweighing the relevance values of components that are non-faulty because both nearest hit and miss transactions are so densely populated they start to resemble each other and this in turn causes many ambiguous results in which many components are identified as faulty.

At 100 and 200 transactions and $h > 0.0$, if we looked into why Ochiai exhibits poor performance at $0.2 \leq lp \leq 0.5$, we observe that the components that are activated the least but have the most involvement in failed transactions are ranked highest, and as a result usually the wrong components are highly ranked or there are many ambiguous results. In the same case, looking at Relief, the correctly ranked components are consistently involved in the nearest miss transactions and hardly in the hit ones which explains why the components will end up with a higher likelihood of being marked as faulty. In the situations where Ochiai is better than Relief from $lp \approx 0.5$, the highest ranked components also have the highest *LexicoRat* values ordered lexicographically. Concurrently, many components or

the wrong components are highly activated in the nearest miss and hit transactions causing incorrect or ambiguous diagnoses in Relief.

For single faults simulated until error detection, Ochiai constantly yields diagnosis results that are superior to those of its counterpart, Relief. Again here the components with the highest *LexicoRat* values ordered lexicographically are correctly identified as faulty. In Relief, the transactions that lead to error have no corresponding nearest hit transactions from the same failed class since there is only a single transaction with a failed class and the faulty component is always involved in it. Moreover, the faulty component is rarely activated as most of its activations are likely to cause an error and eventually terminate the simulation. As a result this causes the decrease in the relevance values of the faulty components and intrinsically its suspicion ranking in Relief.

6.2 Multiple faults

6.2.1 Two Faults

For all types of transactions at $h = 0.0$, Relief performs reasonably better in comparison to Ochiai. In Ochiai the topmost ranked components are also the top two highly placed lexicographically based on their *LexicoRat* values. The problem with that is that components that are highly activated and involved in the same failed transactions as components with low activation are usually ranked higher. This implies that faulty components with lesser activations than others will be lowly ranked resulting in incorrect or ambiguous diagnoses. While for Relief components highly involvement in either nearest miss or hit but not in both increments the relevance weights of components. In the cases where Ochiai is more effective at $lp < 0.4$, the matrices in those results are sparse and this causes the faulty components to be identified correctly because they are likely to be involved in the few failed transaction outcomes in the error vector. Thus, the spectra for these components resemble the error vector the most. On the other hand, Relief at most identifies 1 out of 2 faulty components and the an unidentified faulty component is often hardly involved in the nearest miss and hit transactions implying a low relevance weight and consequently a low suspicion ranking.

At 25 transactions and $h > 0.0$, with lower probabilities $0.2 \leq lp \leq 0.5$ and $0.2 \leq h \leq 0.4$ Ochiai is better than Relief and the two highly ranked components have spectra that highly resemble the error vector. Due to low probability as explained in Observation 6.1.3 the spectra of the faulty components are notably bound to resemble the error vector. Relief is only capable of locating in most cases 1 out of the 2 faulty components, and the correctly identified component is usually the one that is most activated and has its most activations leading to failed transactions. If you look in the nearest miss and hit transactions of Relief, you will always find that this component is highly activated in the miss transactions which explains why its relevance weight is high. For the other component that is not correctly identified, it is always the one with fewer activations and its miss transactions are bound to be full of zeros leading to low relevance weight.

As lp increases while h is kept the same the frequency at which the components are involved in transactions increases and at low h values more transactions lead to more failed transaction outcomes. So for Ochiai this means that many components have spectra that re-

semble the error vector and one of the two faulty components might not be ranked as one of the top two highest components. For Relief since it is more sensitive to the components with most activations in relations to their involvement in failed transaction outcomes, it performs well under this conditions. For those results where Relief outperforms Ochiai because of higher lp, the components are activated more often and the transactions create more failed outcomes, so more components have spectra that are similar to the error vector and these are mostly non-faulty components. In the other cases, Ochiai is better than Relief because there are a lot of activations and a lot of components involvement in failed transactions and dense error vector, so most of the components get weighed highly causing some components other than the faulty ones to be highly ranked. While Ochiai at the same time does better because the faulty components get ranked higher based on the using the *LexicoRat* values.

With 100/200 transactions at $h \leq 0.4$ and $lp \leq 0.2$, the cases in which Ochiai does better than Relief can be explained by the fact that the components are mostly involved in transactions that resemble the error vector and are, therefore, ranked as highest. The reason is that at low h values usually when a faulty component is involved in a transaction, it is bound to cause a failed transaction outcome. So in this case the faulty components all get high *LexicoRat* values. Due to the sparsity of the matrices, Relief ranks higher those components that are involved in a few activations with most of them being part of the failed transactions. The other cases in which Relief is better compared to Ochiai are caused by the fact that numerous spectra that resemble error vector are generated and consequently components other than the faulty ones maybe inccorectly ranked high.

6.2.2 Three Faults

In most cases the behavior of Ochiai and Relief in three faults systems is similar as in those of two faults. However, in cases where $h \leq 0.4$, Relief outperforms Ochiai, and mostly Ochiai ranks those components with the lexicographically highest *LexicoRat* values the highest which means the components that most resemble the error vector and have the most activations are the ones denoted as suspicious even when they are not faulty.

6.2.3 All components faulty

The error vector is more likely to be densely populated since each transaction might lead to an error. In Ochiai this means more components are ranked high since their spectra will be resembling the error vectors. While in Relief most of the ones that resemble the vector will be densely populated themselves and this allows no variation in opposite class which in turn decreases relevance weights to the point that they might be zero which exonerates them from being faulty.

6.3 Computational Complexity

In this section we compare the time/space complexity of Relief and Ochiai. The Relief method comprises two computation steps. In the first, we compute the relevant weights for the components using Relief algorithm which has a time/space complexity of $O(kNC)$

where N denotes the number of transactions, C the number of components and k is the number of iterations. Since in our case $k = N$, the complexity of the first step is $O(N^2C)$. The most complex operation of this step is the selection of the nearest hit and miss transactions because we have to calculate the distance between each transaction and all the rest which takes $O(C)$ comparisons per a transaction. During the second step which involves the suspicion ranking of the components, $O(C) + O(C) + O(C)$ operations are required for normalization of weights, intermediate ranking and final ranking. As a result the total complexity for the Relief method is $O(N^2C) + O(C) + O(C) + O(C)$. However, since $O(N^2C)$ is dominant the computational complexity of this method is $O(N^2C)$.

In Ochiai, the ranking for each component is computed using its spectrum and the corresponding error vector. $O(N)$ comparison operations are required for each component. Therefore, for a C number of components the complexity is $O(NC)$. Looking at the two methods, Ochiai is absolutely faster than Relief. Table 6.1 illustrates the complexity superiority of Ochiai compared to Relief.

	Diagnosis Duration (in seconds)									
No. of transactions	3132	3513	3288	2874	2991	2397	2387	2329	2351	2784
Ochiai	0	0	0	0	0	0	0	0	0	0
Relief	68	83	73	60	63	45	46	44	112	57

Table 6.1: Computational duration (in seconds) for Ochiai and Relief from 10 random activity matrices.

6.4 Statistical Significance of Experimental Results

To test the significance of our experimental results, we use the Student's t Test [26]. Let us optimistically assume that at their best performance, both Ochiai and Relief can diagnose a system and yield results with a 100% fault localization accuracy all the time. Therefore, the population mean for overall diagnoses will be 100% for each of the two methods. Our experimental results comprise two independent samples for Ochiai and Relief diagnoses and we want to compare each of them with the diagnoses population. Each sample contains 315 individual diagnoses and each system consists of an Ochiai and Relief sample. We will use the Single-Sample t Test and the computation is as follows:

1. **Null and Alternative Hypotheses**
   Null hypothesis: Ochiai/Relief diagnoses mean is not different from population mean.
   $(H_0) : \mu = 100$
   Alternative hypothesis: Ochiai/Relief diagnoses mean is different than the population mean.
   $H_1 : \mu \neq 100$

2. **Alpha Level**
   The alpha level specifies a threshold value used to judge whether a test statistic is statistically significant.
   $\alpha = 0.05$

3. **Degrees of Freedom (df)**
   Our samples are 315 large, i.e. $N = 315$.
   $df = N - 1 = 315 - 1 = 314$

4. **Decision Rule**
   Using $\alpha = 0.05$ and two-tailed Single-Sample t Test, *t critical value* $= 1.9719$.
   Our decision rule is:
   If $t \leq -1.9719$ or if $t \geq 1.9719$, reject H_0.
   If $-1.9719 < t < 1.9719$, fail to reject H_0.

5. **Test Statistic Computation**
   $t = \frac{\bar{x}-\mu}{\frac{s}{\sqrt{N}}}$, where $\bar{x}$, s and N are the mean, the standard deviation and the size of the sample respectively.
   The *t values* for diagnoses from 8 systems used in our experiments are:

Systems	t values	
	Ochiai	Relief
1	-25.197	-23.628
2	-25.924	-27.129
3	-33.039	-31.116
4	-29.471	-26.887
5	-31.278	-29.490
6	-30.246	-30.162
7	-28.243	-28.758
8	-31.333	-30.601

Table 6.2: *t values* for Ochiai and Relief samples from 8 systems used in the experiments.

6. **Results**
   The decisions rule was: *If the t value is less than* -1.9719 *or greater than* 1.9719*, we will reject the null hypothesis. Otherwise fail to reject the null hypothesis.*
   Based on the *t values* in Table 6.2, we reject the *null hypothesis*, H_0.

7. **Conclusion**
   Our conclusions is that the diagnoses mean of Ochiai and Relief from our experimental results is different than the 100% diagnoses mean for general population. Table 6.3 depicts the mean values for the samples per a system. By comparing the mean values of the samples to the population mean (100%), we can conclude that the Relief and Ochiai fault localization performance in our experiments is lower than overall diagnoses performance in general.

Systems	Mean Values	
	Ochiai	Relief
1	39.65	50.33
2	38.36	41.06
3	29.02	28.26
4	32.70	40.66
5	28.89	37.89
6	32.94	34.35
7	31.78	33.0
8	31.78	33.0

Table 6.3: Mean values (percentages) for Ochiai and Relief samples from 8 systems used in the experiments.

Chapter 7

Threats to Validity

This section discusses the threats to validity that can affect our empirical study and the results described in the previous chapters.

7.1 Systems

The systems that we used in our experiments are arbitrarily created and they do not necessarily represent some real existing systems. Consequently, we cannot affirm that reproducing our experiments using existing systems that are identical to one or more of our systems will exhibit comparable conclusions. Another threat to our use of the systems is the number of components and links. Our largest system comprises 16 components and 27 links. However, systems in real life may contain hundreds if not thousands of components and links. So our findings may not be representative for such large systems but they may be a good indication on what to expect from such systems.

7.2 Simulations

As our experiments are solely based on simulations using the SFL-Simulator, performing a case study on a real system may produce empirical results that are not existent in our study. In all our simulations, the inception of the simulation starts from a single point. However, the SFL-Simulator allows having several simulation starting points. Performing our experiments using more than one simulation starting point may yield different results than what we obtained. Furthermore, in experiments with simulations using single faults until error detection, we observed that many simulations produced diagnoses with merely a few transactions, in most extreme cases only a single transaction. Such diagnoses may affect the overall diagnosis performance because a single transaction will probably contain insufficient information to produce a good fault localization diagnosis.

7.3 System granularity

In our work we use system topologies to model the systems at a higher component granularity level, specifically at classes or service-oriented systems level. However, working with functions and statements needs a finer component granularity. It is possible to model functions and statements into system topologies too as a topology is nothing but a structure to hold program constructs and how they communicate or are interconnected with each other. Nonetheless, our experiments and findings can only be generalized for systems with a granularity at the level of classes and services. For systems with a granularity lower than in our work, we cannot confirm that our findings will hold.

7.4 Transactions

From our findings, it is obvious that the number of transactions affect the diagnosis performance of the methods we compared. We have used a maximum number of 200 transactions in our experiments excluding the experiments with singles faults and single error detection because the number of transactions in these cases are simulation-dependent. Using a number of transactions greater than what we experimented with may yield results which we cannot defend in our work. Therefore, our work specially with multiple faults is limited to a maximum of 200 transactions as per a system configuration.

7.5 Faults, Health Probability, Link Probability

In our experiments we only set predetermined components as faulty and not every component in turn. Performing experiments whereby all faulty permutations of components are investigated may yield results different than what we observed. For health and link probability values we also chose certain intervals from 0.0 to 1.0. However, using different intervals may produce results different than ours.

7.6 Relief Algorithm

In cases where the transaction outcomes are all pass or contain a single fail outcome, some transactions may not have a corresponding nearest hit/miss transaction. This condition is not considered by the Relief Algorithm 1. We have circumvented this condition by setting the $\delta(i, X_k, X^h)$ and $\delta(i, X_k, X^m)$ values in Equation 2.9 to zero when computing the relevance weights. Although this applies to all components when calculating their weights, without a thorough validation, we are not completely sure what effect this has on Relief.

Another treat concerning Relief is that we have slightly diverted from the fashion in which normalization is performed as stated in the Ranking Algorithm 2. In the step of normalization $Y[i] \geq 0$ is used, but we use $Y[i] > 0$ instead because we are of opinion that if the value of $Y[i] = 0$ then the corresponding normalized weight will be zero (already handled by the else block in the algorithm) and that extra computation step $Y[i] = 0$ is redundant. Also in the step of translating the weights into intermediate ranks there is a little confusion in

the original paper by Roychowdhury and Khurshid [6], they mention using both the number of transactions and the number of components. However, in their calculations they used the number of transactions which sometimes produces negative scores for lowly ranked components. We have opted for the number of components because it produces positive scores and makes it easier for visual comparison with Ochiai scores. Whichever approach you take the components will still have the same order in the ranking. But this might be a point to take note of when reproducing our work.

In addition to the preceding point, another concern about the normalization step is that it does not encompass negative weights. All negative weights are automatically normalized to zero. There may be a situation whereby all the components have negative relevance weights, this condition is not considered in the normalization step of the Ranking Algorithm 2. So considering normalization of negative weights may have some effect on the overall relevance weights.

7.7 Statistical Significance of Experimental Results

In our computation of t-Test we have aggregated the diagnosis results from all results permutations into a single sample for each system. Creating samples separately for each of these variables may yield tests that are different than what we have obtained. Also for the computation of the *t value*, our sample size is large, 315 with 314 degrees of freedom (df), and cannot be found in most *critical t values* tables. Most *critical t value* tables have a list with *critical t values* corresponding up to 200 df. However, we are allowed to take the nearest biggest df in the table that is smaller than 314, like e.g. 200. This may affect the tests computation slightly and mostly without serious implications.

Chapter 8

Related Work

This chapter is devoted to the related work. Considering the fact that the work in this thesis comprises comparing two different fault localization methods namely Ochiai and Relief, we give each method a related work section. The first section presents studies about Ochiai, whereas the following section discusses previous research in Relief. The last section is for other relevant studies.

8.1 Ochiai

Abreu et al. [4] in their article about the accuracy of spectrum-based fault localization, studied the effectiveness of different SFL similarity coefficients. They presented results that demonstrated that for the Siemens set with single-site faults, Ochiai has improved diagnostic accuracy over other eight SFL methods. In another study by Abreu et al. [8] where they researched using Siemens set in the utility of low-cost, generic invariants known as screeners to detect errors in online SFL, the results showed how screeners can be used in online diagnosis with reasonable overhead and also Ochiai was repeatedly seen to outperform Tarantula and Jaccard. The toolset, Zoltar, talked about in the study by Janssem et al. [11] which is used for automatic fault localization also produced diagnostic results that illustrate that Ochiai is better than its counterparts. Naish et al. [12] used several coefficients to evaluate their SFL approach and for single-fault programs their results demonstrated that Ochiai performed better than the rest. Wang et al. [27] used 22 different SC measures including Tarantula and Ochiai in their approach to build composite measures. The composite measure performed better than the constituent SC measures. Yu et al. [28] and Piel et al. [2, 9] used Ochiai in their respective methods for SFL diagnosis all producing promising results. Bandyopadhyay [29] described in his article an approach that enhances effectiveness of SFL by incorporating the relative importance of different test cases to compute the SC scores. The paper stated that SFL techniques such as Tarantula and Ochiai consider each test case equally important with respect to SC scores computation. However, according to Bandyopadhyay, research on test case generation and selection techniques has indicated that utilizing a select number of test cases can enhance the effectiveness of fault localization, in other words its SC scores. In his proposed approach, Bandyopadhyay used a SC which

is an adaptation of Ochiai. Based on his work, the new SC had a slight improvement in effectiveness in comparison to Ochiai. In their paper about spectrum-based multiple fault localization, Abreu et al. [30] presented new framework known as BARINEL in which a program is modeled using abstractions of program traces (as in SFL) while Bayesian reasoning is used to deduce multiple-fault candidates and their probabilities. According to their experimental results BARINEL typically outperformed current SFL approaches such as Ochiai at a cost complexity that is only marginally higher.

Generally, the work in this thesis differs from the above mentioned studies in that we model the systems under investigation with larger component granularity and use that for the experiments instead of test sets such as the Siemens set. Furthermore, we study the effect of multiple faults in our experiments which is hardly the case in most of the aforementioned studies. Also, none of the studies mentioned have compared Ochiai to Relief.

8.2 Relief

Roychowdhury et al. [6] performed a study to apply Feature Selection based Relief technique to SFL, and compared the diagnosis performance of Relief with Ochiai and other algorithms. In their research they only assessed those algorithms in single-fault cases with the Siemens Test Suite, while our work explores the diagnostic performance of Ochiai and Relief for both single and multiple faults in several different systems. In addition, the assessment metrics they used are different than the ones we employ. They check the percentages of examined code before the fault is localized for each algorithm, whereas we compare how many times the algorithms are able to correctly and uniquely identify the faulty component(s).

In other studies about Relief: Guyon et al. [17], Kira et al. [18], Hum et al. [31], Sunet et al. [20] and Blum et al. [19] all talk about Feature Selection algorithms including Relief. In some of these articles, Relief has been acclaimed as a good algorithm for datasets with many features. Our works differs from these papers based on the fact that they only consider Relief without any mention of fault localization.

8.3 Others

Cotroneo et al. [32] addressed problems of online software fault diagnosis in complex safety critical software systems. Their approach combined both error detection and fault location processes. Yu et al. [33] in their paper, *Towards Practical Debugging for Regression Faults*, proposed an automated technique to locate failure-inducing changes based on Delta Debugging, which systematically searching for failure-inducing changes by patching a subset of changes and observing execution results. Ali et al. [34] developed a new program called Concordance for conducting fault localization experiments. They evaluated several fault localization techniques and according to them Tarantula performed well. In another survey about fault localization done by Wong et al. [35], they investigated program spectrum-based, static dynamic and execution sliced-based, machine learning-based and model-based and data mining-based methods. Debroy et al. [36] talked about the equivalence of some

fault localization techniques in their paper. Their evaluation approach defines equivalence relation as the virtue by which two or more fault localization techniques may be considered equivalent if they produce identical rankings of program components. Abreu et al. [37] proposed a framework known as DEPUTO to improve the effectiveness of SFL techniques. Their approach combines spectrum-based fault localization with a model-based debugging approach which refines the ranking obtained from the spectrum-based method by filtering out those components that do not explain the observed failures when the program semantics is considered. Perez et al. [13] discussed in their paper a lightweight, topology-based analysis approach to diagnose a system based on the source code structure. The approach uses both SFL and Dynamic Code Coverage (DCC) techniques and the approach makes a choice which fault localization technique to use by creating an hierarchical model of the system.

Again, our work differs from these studies because they have not considered fault localization using Relief and in most cases have no mention of Ochiai or the comparison of the two methods.

Chapter 9

Conclusions and Future Work

To bring this thesis to a close, we summarize the reflections on our findings and draw some conclusions. Finally, we present some ideas for future work.

9.1 Conclusions

In this thesis, we have compared the performance of two spectrum-based fault localization techniques namely Ochiai and Relief. Ochiai stems from statistical approaches and uses a similarity coefficient to order the components of a system according to their likelihood of being faulty, while Relief as a machine learning approach uses feature selection to narrow down the search space for the components that are likely to be faulty by assigning them higher relevance weights.

To address the research questions in this thesis, several experiments have been conducted with different systems and system configurations using the SFL-Simulator (Chapter 4 and 5). The research questions as stated in the introduction of this thesis have been answered thoroughly through the evaluation and interpretation (Chapter 6) of the experimental results (Chapter 5). We have observed that there is no clear-cut winner in regard to the comparison of the performance of the two methods. The diagnosis effectiveness does not only depend on a fault localization method but also on many other factors such as the system under investigation, the components in the system and links between them, the number of faults in the system, the health state of the components, the link probabilities and the number of transactions used. The summation of the answers to the research questions are as follows:

Research Question 1:
Considering the two techniques Ochiai and Relief: Which one has a better fault localization performance or accuracy?

Systems with Single Faults In systems with predetermined number of transactions and single faults with no fault intermittency[1], i.e. $h = 0.0$ for faulty components, both Ochiai and

[1] Fault intermittency is when the health value of the faulty component is $h > 0.0$.

Relief exhibited identical diagnosis performance. Using very few transactions (≤ 25) with high fault intermittency ($h > 0.0$) and low link probability between the components, Relief performed reasonably well in comparison to Ochiai. However, with high link probability values Ochiai was more effective than Relief under fault intermittency. With a high number of transactions (≤ 200) and fault intermittency, Ochiai showed poorer performances than Relief at low link probability values. As the link probability increased Ochiai performed better than Relief. In cases whereby the system simulations were run until an error was detected, Ochiai repeatedly outperformed Relief.

Systems with Multiple Faults In systems with up to three faults without fault intermittency, in most cases Relief had a better fault localization diagnosis accuracy than Ochiai. In some cases under the same condition, Ochiai was more effective than Relief with low link probability. With fewer transactions, low fault intermittency ($h \leq 0.4$) and link probability ($lp \leq 0.5$) Ochiai is better than Relief. But under high link probability in some cases Relief performed well in comparison to Ochiai. When a large number of transaction is used with fault intermittency in most cases Ochiai did better than Relief under low link probability values combined with low fault intermittency. When all the components are set as faulty, Ochiai clearly produced better fault localization performance than Relief.

Research Question 2:
Are there characteristics of systems that support the application of: Ochiai or Relief?

The most important characteristics of a system that affect the performance of both Ochia and Relief are: the number of faults, the links between the components of the system and the number of transactions executed by the system. These characteristics are not mutually exclusive and their combined effect is what influences the fault localization accuracy of the two techniques. The influence of these factors have already been discussed thoroughly in our discussions in Chapter 6 and in the summation of the answers to **Research Question 1**.

In regard to the performance of Ochiai and Relief, another factor that is important in the comparison of the two methods is computational complexity. It is not only important to consider the fault localization accuracy of a method but also the time and space efficiency by which the method performs its diagnosis. The results show clearly that Ochiai is faster than Relief in all cases investigated.

9.2 Future work

The results attained in this thesis are promising and are a good step towards finding and choosing the appropriate diagnosis technique based on the characteristics of the system under investigation. Based on the results of our study, we suggest a number of possible future works:

- Performing a study of comparison of Ochiai with other variants of the Relief Algorithm such as Relief-(A to F).

- Performing a study similar to ours using systems consisting of larger numbers of components and links, maybe in the order of hundreds as well as larger numbers of transactions.
- Performing our study but with multiple faults using single error detection simulations.
- Adding automatic system topology generation features to the SFL-Simulator.

Bibliography

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault localization technique," *In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282, 2005.

[2] E. Piel, A. Gonzalez-Sanchez, H.-G. Gross, A. J. van Gemund, and R. Abreuy, "Online spectrum-based fault localization for health monitoring and fault recovery of self-adaptive systems," *ICAS 2012 : The Eighth International Conference on Autonomic and Autonomous Systems*, pp. 64–73, 2012.

[3] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," *4th Symposium on Search Based Software Engineering*, 2012.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," *TAICPART-MUTATION '07 Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp. 89–98, 2007.

[5] K. Yin, R. Rengaswamy, S. N. Kavuri, and V. Venkatasubramanian, "A review of process fault detection and diagnosis part i: Quantitative model-based methods," *Computers and Chemical Engineering*, pp. 293–311, 2003.

[6] S. Roychowdhury and S. Khurshid, "Software fault localization using feature selection," *Proceeding MALETS '11 Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, pp. 11–18, 2011.

[7] R. Abreu, P. Zoeteweij, and A. J. V. Gemund, "A new bayesian approach to multiple intermittent fault diagnosis," *Proceeding IJCAI'09 Proceedings of the 21st international jont conference on Artifical intelligence*, pp. 653–658, 2009.

[8] R. Abreu, A. Gonzalez, P. Zoeteweij, and A. J. van Gemund, "Automatic software fault localization using generic program invariants," *SAC '08 Proceedings of the 2008 ACM symposium on Applied computing*, pp. 712–717, 2008.

[9] E. Piel, A. Gonzalez-Sanchez, H.-G. Gross, and A. J. van Gemund, "Spectrum-based health monitoring for self-adaptive systems," *Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on*, pp. 99–108, 2011.

[10] R. Abreu, A. Gonzalez-Sanchez, and A. J. van Gemund, "Exploiting count spectra for bayesian fault localization," *PROMISE '10 Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pp. –, 2010.

[11] T. Janssem, R. Abreu, and A. J. van Gemund, "Zoltar: A toolset for automatic fault localization," *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 662–664, 2009.

[12] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *2011 ACM Transactions on Software Engineering and Methodology*, 2011.

[13] A. Perez, A. Riboira, and R. Abreu, "A topology-based model for estimating the diagnostic efficiency of statistics-based approaches," *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium*, pp. 171–176, 2012.

[14] SERG-Delft, "Spectrum-based fault localization (sfl) simulator." `https://github.com/SERG-Delft/sfl-simulator`, 2013.

[15] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," *In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 1998)*, pp. 83–90, 1998.

[16] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. J. van Gemund, "An empirical study on the usage of testability information to fault localization in software," *SAC '11 Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1398–1403, 2011.

[17] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, pp. 1157–1182, 2003.

[18] K. Kira and L. A. Rendell, "A practical approach to feature selection," *Proceeding ML92 Proceedings of the ninth international workshop on Machine learning*, pp. 249–256, 1992.

[19] A. L. Blum and P. Langley, "Selection of relevant features and examples in machine learning," *Artificial Intelligence - Special issue on relevance*, pp. 245–271, 1997.

[20] Y. Sun and J. Li, "Iterative relief for feature weighting," *Proceeding ICML '06 Proceedings of the 23rd international conference on Machine learning*, pp. 913–920, 2006.

[21] A. Smola and S. Vishwanathan, *Introduction to Machine Learning*. Cambridge University Press, 2008. (ISBN 0 521 82583 0).

[22] Ruby, "Ruby version 1.9.3." `http://www.ruby-lang.org/en/downloads` .

[23] GraphViz, "Graphviz." http://www.graphviz.org .

[24] RubyGems, "Ruby-graphviz gem." http://rubygems.org/gems/ruby-graphviz .

[25] RubyGems, "Colorize gem." http://rubygems.org/gems/colorize .

[26] Wikipedia, "Student's t-test." https://en.wikipedia.org/wiki/Student's_t-test .

[27] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," *ASE '11 Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 556–559, 2011.

[28] K. Yu, M. Lin, Q. Gaoy, H. Zhangy, and X. Zhangy, "Locating faults using multiple spectra-specific models," *SAC '11 Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1404–1410, 2011.

[29] A. Bandyopadhyay, "Improving spectrum-based fault localization using proximity-based weighting of test cases," *ASE '11 Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 660–664, 2011.

[30] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-based multiple fault localization," *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 88–99, 2009.

[31] Q. Hu, P. Zhu, J. Liu, Y. Yang, and D. Yu, "Feature selection via maximizing fuzzy dependency," *Fundamenta Informaticae*, pp. 167–181, 2010.

[32] D. Cotroneo, G. Carrozza, and S. Russo, "Software faults diagnosis in complex ots based safety critical systems," *Seventh European Dependable Computing Conference*, pp. 25–34, 2008.

[33] K. Yu and M. Lin, "Towards practical debugging for regression faults," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 487–490, 2012.

[34] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," *ASE '09 Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 76–87, 2009.

[35] W. E. Wong and V. Debroy, "Survey of software fault localization," *Technical Report UTDCS-45-09*, 2009.

[36] V. Debroy and W. E. Wong, "On the equivalence of certain fault localization techniques," *SAC '11 Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1457–1463, 2011.

[37] R. Abreu, W. Mayer, M. Stumptner, and A. J. van Gemund, "Refining spectrum-based fault localization rankings," *SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 409–414, 2009.

Appendix A

Ruby Implementation of Relief

```
# Retrieves nearest hit and nearest miss for a given observation/row
# Param(s) => index: the index of the observation
# observations: list of list of observations
# error: error vector
# Return =>  hit: nearest hit (observation)
# miss: nearest miss (observation)
def retrieve_neighbors(index, observations, error)
  sample = observations[index]
  hitHash = Hash.new
  missHash = Hash.new

  observations.each_index do |i|
    next if index == i
    neighbor = observations[i]
    euclideanDistance = retrieve_euclidean_distance(sample, neighbor)
    if error[index] == error[i]
      hitHash[i] = euclideanDistance
    else
      missHash[i] = euclideanDistance
    end
  end

  # Sort and take hit the with the first/with smallest distance
  hitArr = hitHash.sort_by{|k,v| v}.first
  # Sort and take the miss with the first/with smallest distance
  missArr = missHash.sort_by{|k,v| v}.first
  hit = (hitArr.nil?) ? [] : observations[hitArr.first]
  miss = (missArr.nil?) ? [] : observations[missArr.first]
  return hit, miss
end
```

```
# Retrieves the Euclidean distance of two lists
# Param(s) => array1: a list or observation
# => array2: a list or observation
#
# Return => Euclidean distances between the two lists
def retrieve_euclidean_distance(array1, array2)
  distance = 0
  array1.each_index do |i|
    distance += (array2[i] - array1[i])**2
  end
  return Math.sqrt(distance)
end
```

```
# Normalizes components' relevant weights in the range of 0.0..1.0
# Param(s) => weights: list of relevant weights for components involved
#
# Return => normWeights: list of normalized relevant weights for components involved
def normalize_weights(weights)
  normWeights = Array.new(weights.length); maxValue = weights.max
  weights.each_index do |i|
    value = weights[i]
    if value > 0.0
      normWeights[i] = value / maxValue
    else
      normWeights[i] = 0.0
    end
  end
  return normWeights
end
```

```
# Normalizes components by ranking them in range of 1.0..nr of components involved
# Param(s) => norm_weights: list of normalized relevant weights for components
#
# Return => ranked_weights: list of relevant weights for components involved
def translate_weights_to_ranks(norm_weights)
  rankedWeights = Array.new(norm_weights.length)
  lines = norm_weights.length
  norm_weights.each_index do |i|
    value = norm_weights[i]
    rankedWeights[i] = (value + lines * (1 - value)).floor
  end
  return rankedWeights
end
```

```
# Retrieves relevant weights for components/links using RELIEF Algorithm
# Param(s) => observations: list of observations/tests
# error: list of observation outcomes
#
# Return => a list with relevant weights for each component/link
def retrieve_relief_weights(observations, error)
  rows = observations.length
  cols = observations[0].length
  weights = Array.new(cols, 0.0)
  for i in 0..(rows - 1)
    hit, miss = retrieve_neighbors(i, observations, error)
    hitLength = hit.length; missLength = miss.length
    for j in 0..(cols - 1)
      obsValue = observations[i][j].to_f
      diffHit = (hitLength == 0) ? 0 : ((obsValue - hit[j].to_f)**2)
      diffMiss = (missLength == 0 ) ? 0 : ((obsValue - miss[j].to_f)**2)
      weights[j] = weights[j] + (diffMiss - diffHit) / rows.to_f
    end

  end

  # Normalize weight to the range of 0.0..0.1
  normWeights = normalize_weights(weights)
  # Rank weights to the range of 1..nr of components
  rankWeights = translate_weights_to_ranks(normWeights)
  return rankWeights
end
```

```
# Computes the similarity/relevance score of the components based on the fact
# whether they are faulty or not
# Param(s) => observations: list of observations/tests
# error: list of observation outcomes
# comps: list of components' names
#
# Return => a list components with their corresponding relevance scores
def compute_relief_scores(observations, error, comps)
  scores = Hash.new
  weights = retrieve_relief_weights(observations, error)
  linesCount = weights.length
  maxVal = weights.max
  maxCardinality = weights.count(maxVal)
  weightFactor = (linesCount + 1.0 - maxCardinality) / linesCount.to_f
  weights.each_index do |i|
```

```
    scores[comps[i]]=(((1-(weights[i]/linesCount.to_f))*weightFactor.to_f)).round(3)
  end
  return scores.sort_by{|k,v| v}.reverse
end
```

Appendix B

Systems

Figure B.1: System 1 with 6 components and 7 links.

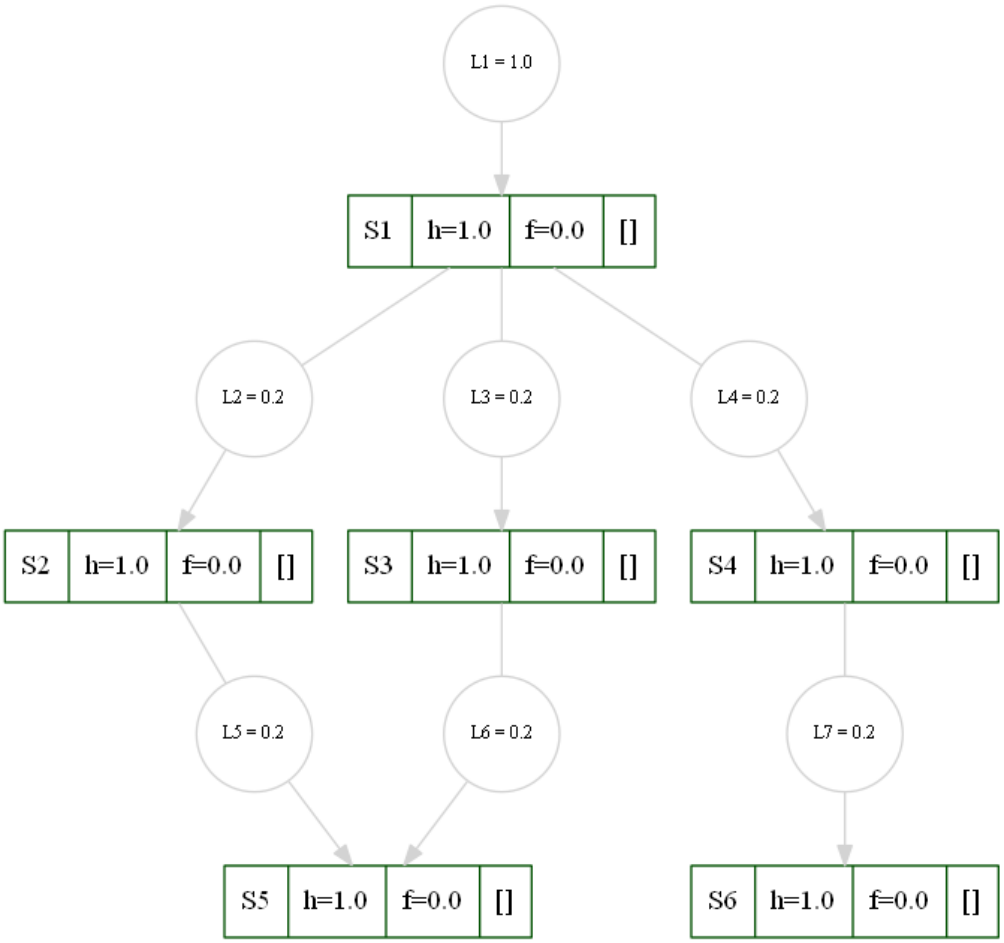

Figure B.2: System 2 with 6 components and 11 links.

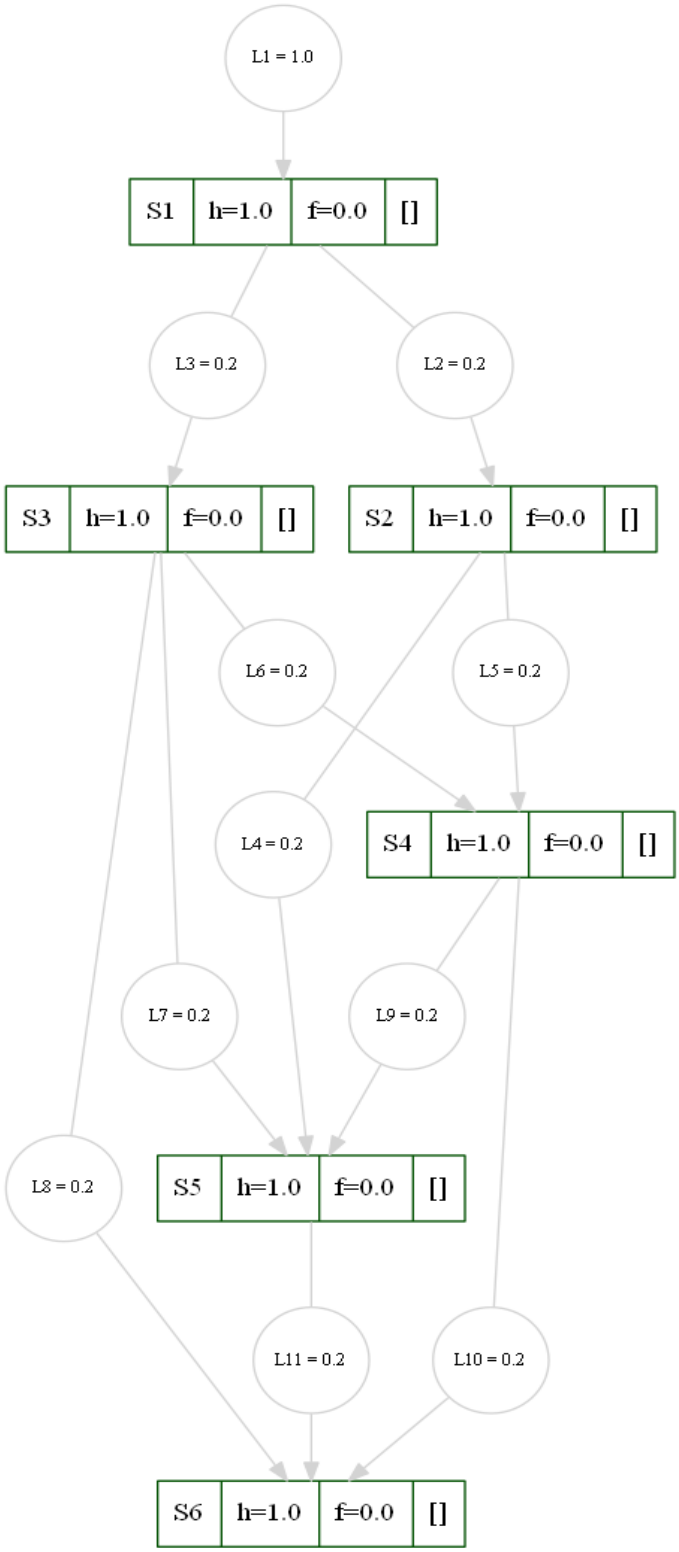

Figure B.3: System 3 with 8 components and 7 links.

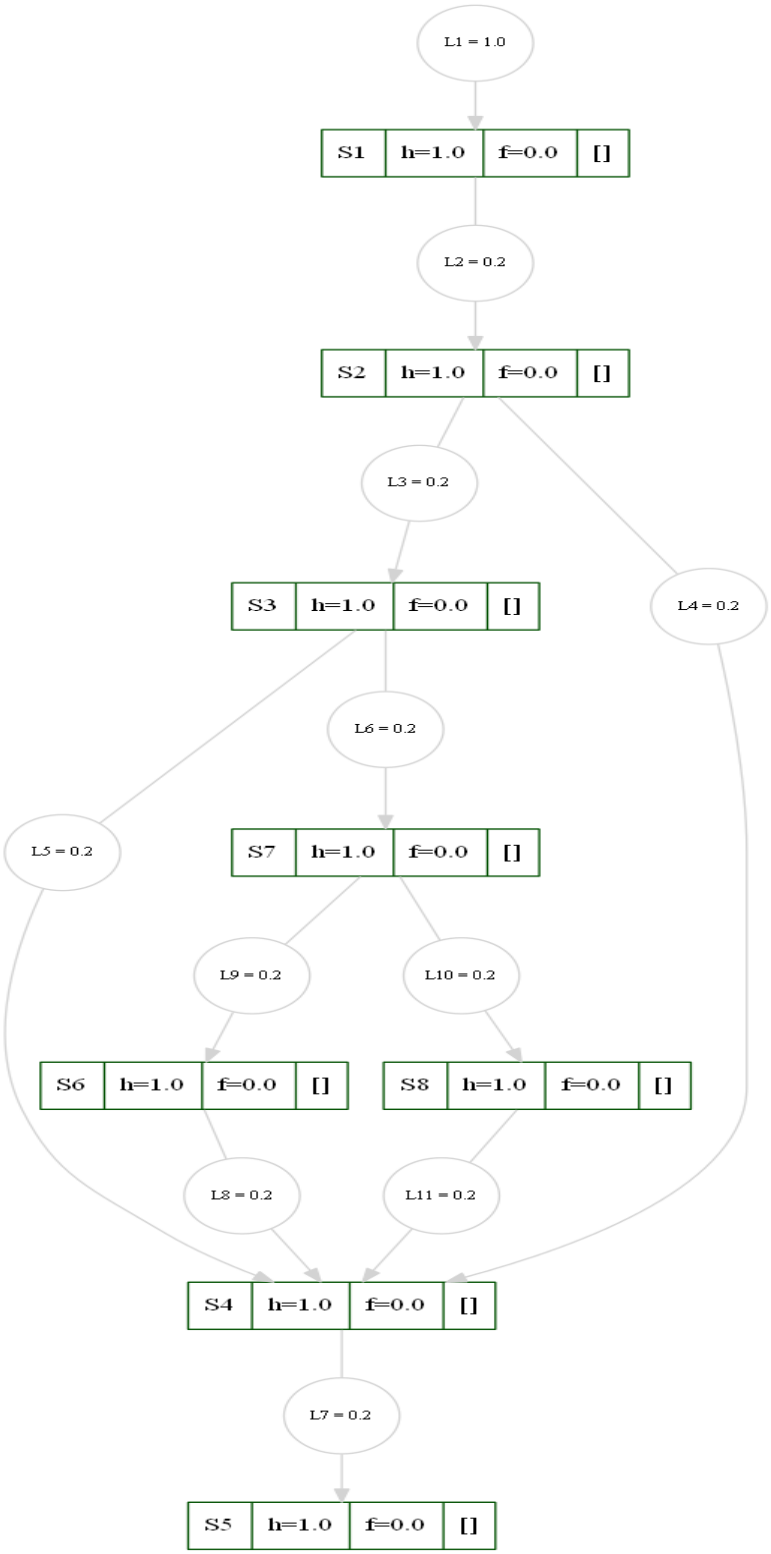

Figure B.4: System 4 with 18 components and 21 links.

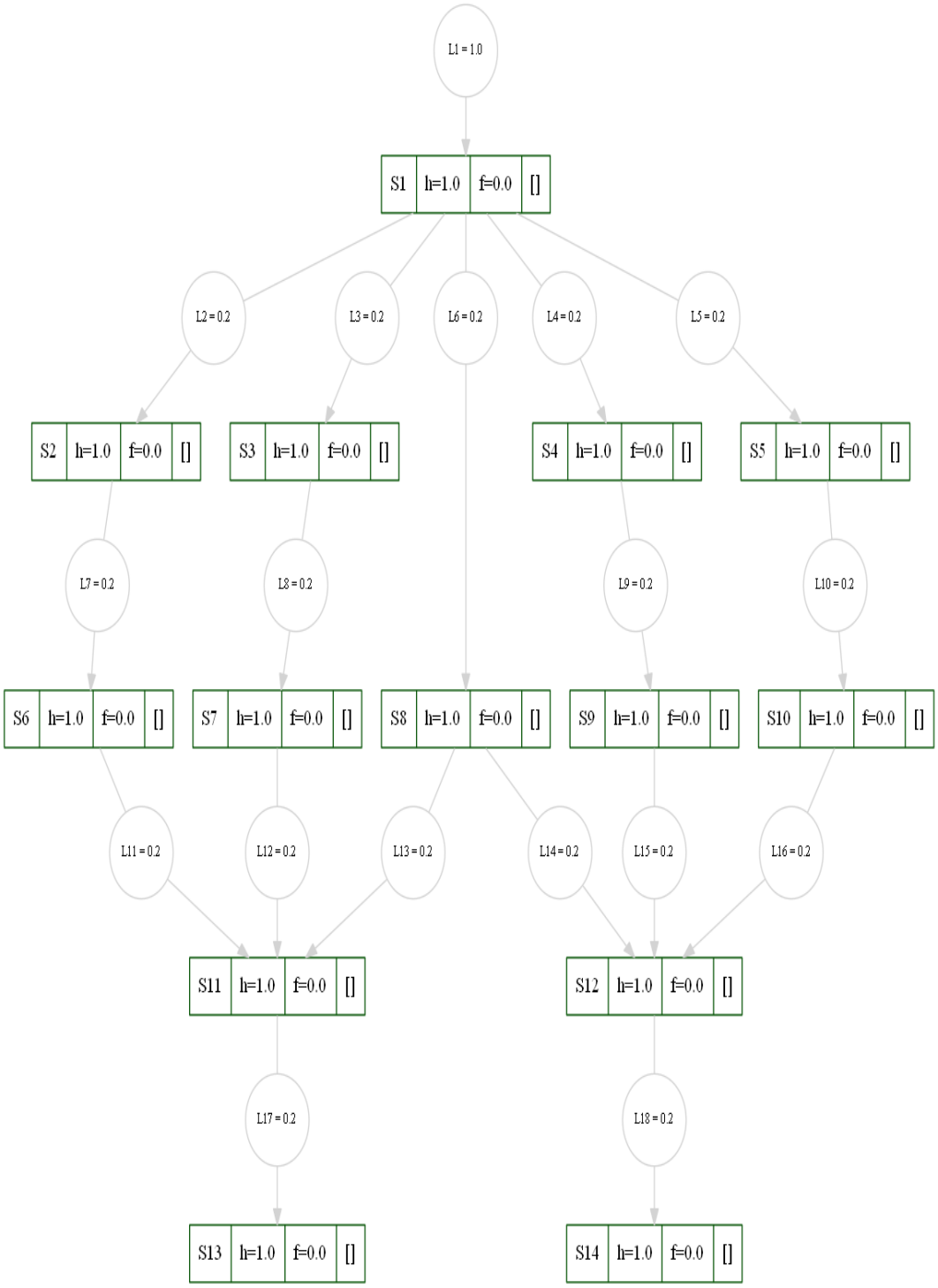

Figure B.5: System 5 with 8 components and 16 links.

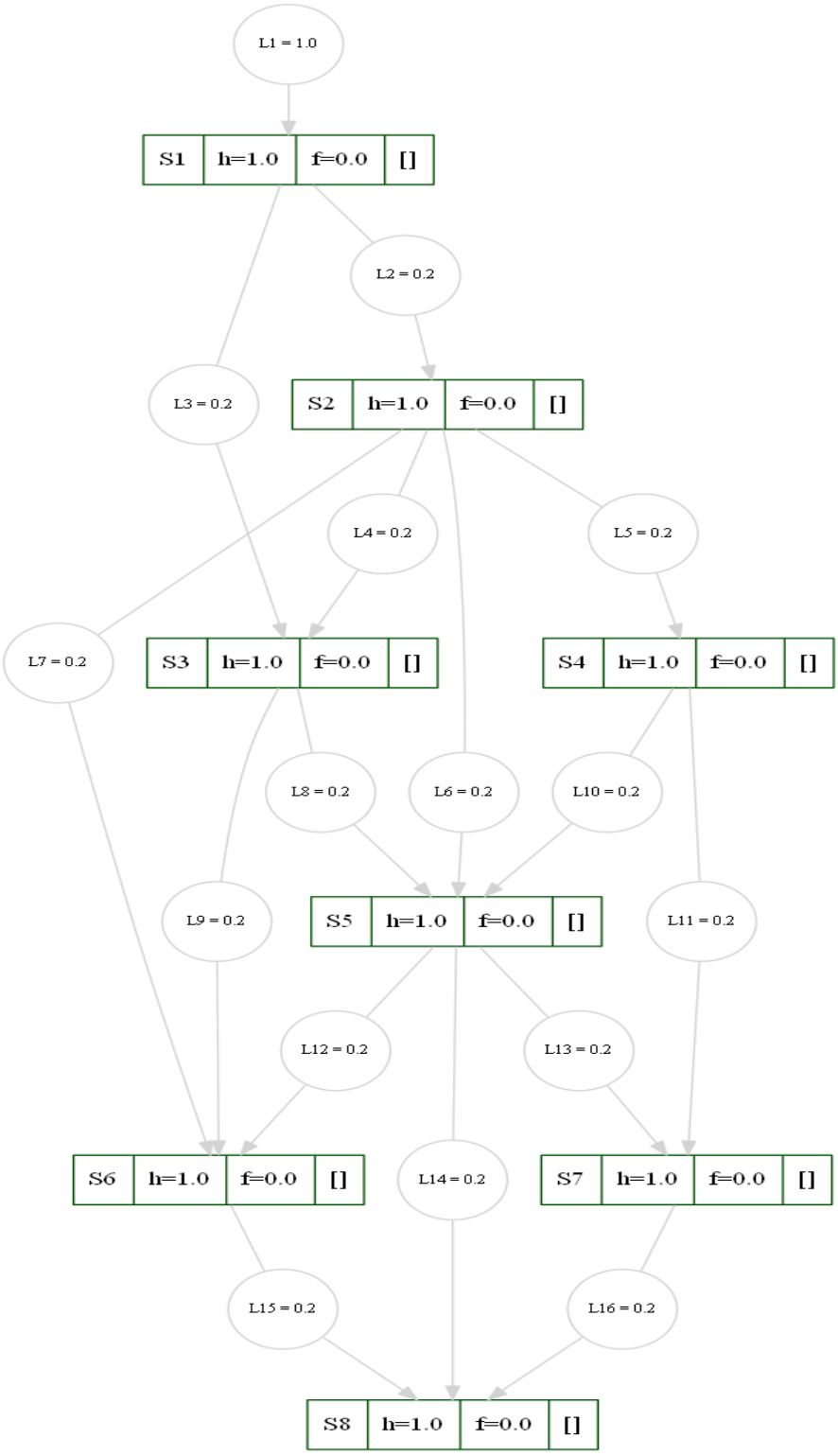

Figure B.6: System 6 with 16 components and 27 links.

Figure B.7: System 7 with 8 components and 11 links.

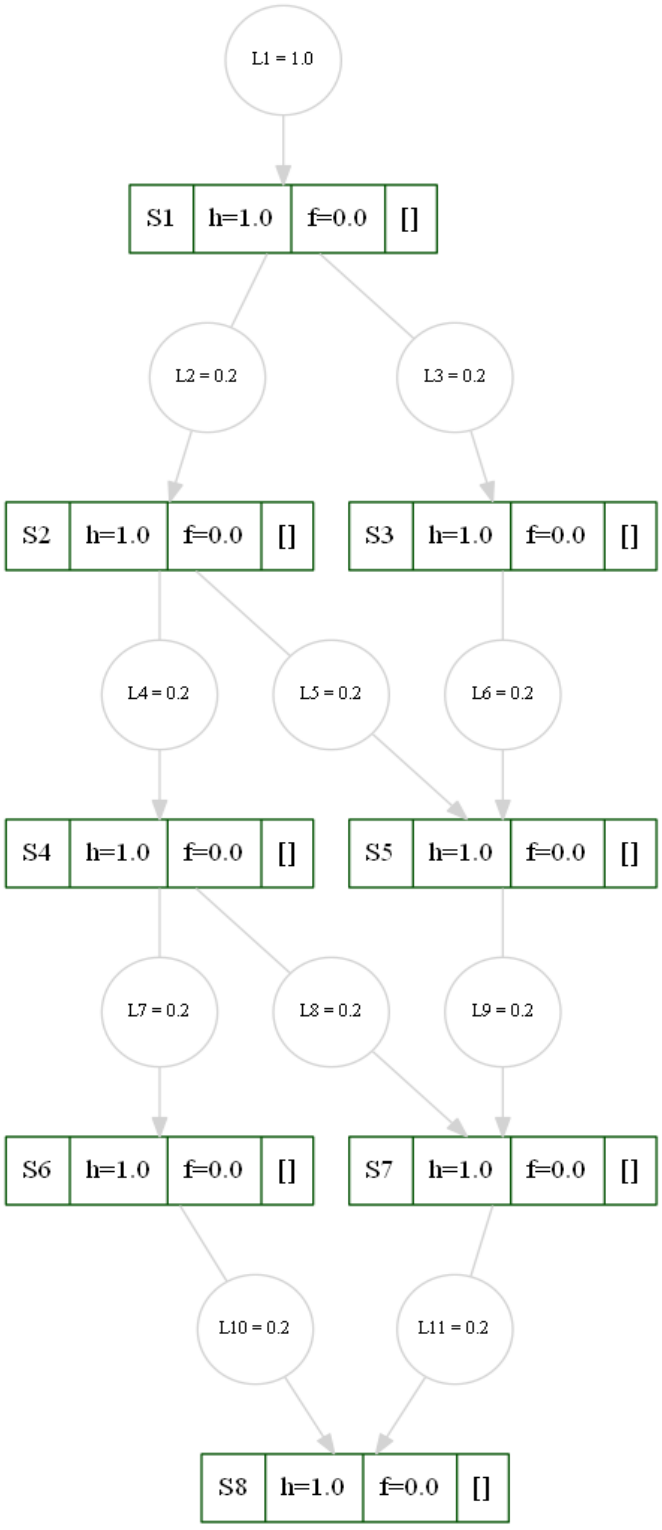

Figure B.8: System 8 with 8 components and 21 links.