August 2013

Localizing State-Dependent Faults Using Associated Sequence Mining

Shaimaa Ali
The University of Western Ontario

Supervisor
Jamie Andrews
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

LOCALIZING STATE-DEPENDENT FAULTS USING ASSOCIATED SEQUENCE MINING

(Thesis format: Monograph)

by

Shaimaa Ali

Graduate Program in Computer Science

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

Abstract

In this thesis we developed a new fault localization process to localize faults in object oriented software. The process is built upon the "Encapsulation" principle and aims to locate state-dependent discrepancies in the software's behavior. We experimented with the proposed process on 50 seeded faults in 8 subject programs, and were able to locate the faulty class in 100% of the cases when objects with constant states were taken into consideration, while we missed 24% percent of the faults when these objects were not considered. We also developed a customized data mining technique "Associated sequence mining" to be used in the localization process; experiments showed that it only provided slight enhancement to the result of the process. The customization provided at least 17% enhancement in the time performance and it is generic enough to be applicable in other domains. In addition to that we have developed an extensive taxonomy for object-oriented software faults based on UML models. We used the taxonomy to make decisions regarding the localization process.It provides an aid for understanding the nature of software faults, and will help enhance the different tasks related to software quality assurance. The main contributions of the thesis were based on preliminary experimentation on the usability of the classification algorithms implemented in WEKA in software fault localization, which resulted in the conclusion that both the fault type and the mechanism implemented in the analysis algorithm were significant to affect the results of the localization.

Keywords: software fault localization, data mining, software faults taxonomy

Acknowlegements

Thanks to my supervisor ”Dr. Jamie Andrews”, advisors ”Dr. Nazim Madhavji” and ”Dr. Robert Mercer” and examiners ”Dr. Hanan Lutfiyya”, ”Dr. Michael Bauer”, ”Dr. David Bellhouse” and ”Dr. Carson Leung” for the help and advice they provided for the work in this thesis.

Very special thanks to my parents and siblings for their support. ”I wouldn’t have done it without you”

Thanks to my friends who gave me the social support and became like a family for me while I’m all alone in Canada.

Many thanks to my fellow Egyptians, my fellow Muslims and my fellow humans who supported me during my crisis without even knowing how I was. ”You have restored my faith in humanity and I look forward to give back anonymously just like you did to me”

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Maintenance is one of the most costly tasks of software engineering, especially when the program grows in size and complexity. A central process of software maintainance is the process of testing and debugging. The National Institution of Standards and Technology (NIST) estimated that software errors cost the economy of the United States about $59.5 billion annually; that amount represents about 0.6 % of the gross national product of the United States in 2002 [22]. Therefore, dealing with software errors (also known as bugs) through testing and debugging is one of the important tasks in software development. In addition to that, previous research indicated that approximately 50% of the cost of new systems development goes to testing and debugging [48], let alone the cost of maintenance during the lifetime of the software.

As described in Figure 1.1, the testing and debugging process starts with a subject program that contains a number of faults, usually referred to as bugs. A test suite is prepared in order for the testing process to begin. The result from the testing process contains the execution behavior of each test case as well as whether it was successful or failing. This information is then used for predicting the components of the subject program containing the faults that caused the failures during the fault localization process. The suspected components are then evaluated to determine whether they actually contained the fault. If so, the correction process begins, aiming towards correcting the discovered fault(s), and the desired output is the subject

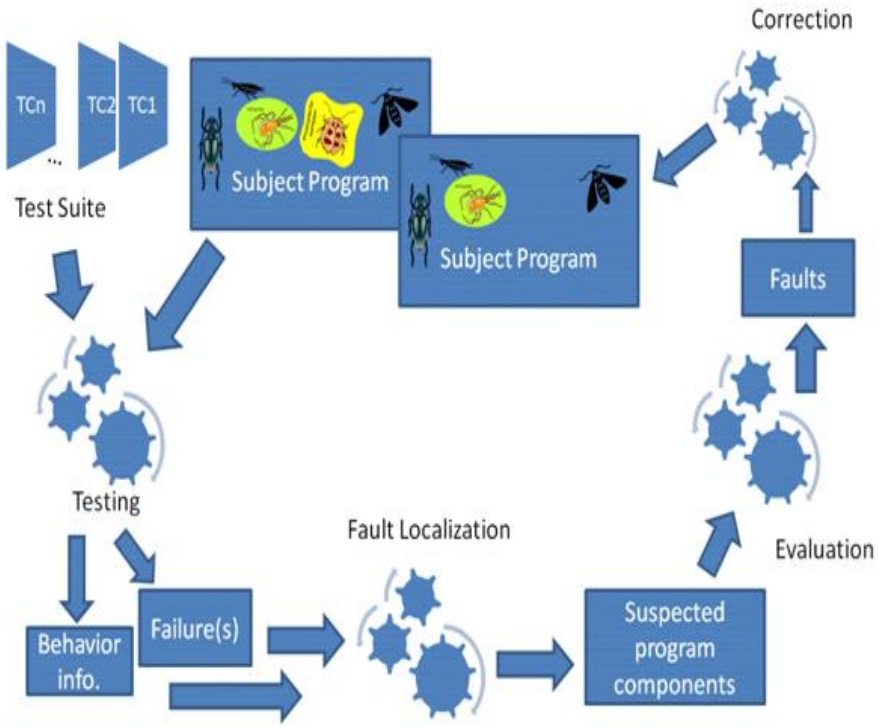

Figure 1.1: Testing and debugging process

program with a reduced number of faults in it. The scope of research of this thesis is on the fault localization step. More specifically we are focused fault localization in object oriented software programs.

1.1 Fault Localization

The fault localization problem is the problem of finding the location in the source code that caused a known failure by analyzing the behavior of the program during successful and failing test cases.

Researchers have spent a great amount of effort to automate the tasks involved in the testing and debugging process in order to reduce the cost associated with them. One of the most challenging tasks of the process is fault localization; i.e., after a failure has been detected, the root cause of that failure needs to be identified then fixed.

Unfortunately, even though some of the automated fault localization techniques are theoretically very promising, they have not been practically used by developers. In 2011 Parnin and Orso [45] examined whether automated software debugging is helpful for programmers. In particular, they studied the usability of Tarantula [32] for programmers, as it is similar to most of the state-of-the-art techniques and it was found to be the one of the most effective among them [31]. They concluded that the programmers do not visit the ranked lines in order, and that they prefer to have information about higher-level constructs as they can judge their relevance to the failure using their names. In addition, Thung et. al [37] examined the possibility of localizing hundreds of faults in three different Java programs, namely, AspectJ, Rhino and Lucene. They concluded that faults are not localizable in a small number of lines, while 88% of the faults they examined were localizable in one or two files.

1.2 Data Mining

One of the major challenges that faces fault localization techniques is the explosion of data [7]. For complex programs the amount of data that needs to be analysed to determine the faulty component can be enormous.

As defined by [27], data mining is the analysis of large observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner; these relationships and summaries are referred to as models or patterns.

From its definition, data mining seems to be the solution for the data explosion problem. In addition to that, the intelligent analysis provided by data mining techniques would allow for more sophisticated solutions for the fault localization problem.

1.3 Thesis Contribution

We started the work on this thesis with exploratory experimentation on the usability of already existing data mining techniques, more specifically, classification techniques implemented in the Weka tool [1], in the problem of fault localization. We found that both the mechanism implemented in the classification technique and the nature of the fault to be localized affected the accuracy of the results significantly.

Combining the findings of Parnin and Orso [45] with those of Thung et al. [37] and our initial findings, we aimed at developing a customized algorithm for fault localization that works at the class level, that is, localizes the faulty class in object oriented software, taking the nature of the faults to be localized in consideration.

In order to achieve that goal, we needed to understand the different types of faults that can occur in object-oriented software. There are several taxonomies of software faults in the literature; however, none of them provided the answer for the question we were looking at. That is, how can we take the nature of the fault in consideration while trying to localize it? Therefore, we created a new fault taxonomy based on the artifacts that appear in different UML diagrams. Based on our study of the different types of faults, we decided to target one specific type, namely, state-dependent faults, as it allows us to indirectly target the other types as well.

Following that decision, we developed a process to collect and analyse information from the subject program that aims at finding inconsistencies in the state-dependent behavior of that program. We also created a customized mining algorithm that combines two existing mining techniques, namely, association mining and sequence mining, that we called "associated sequence mining", to be used during this localization process. Finally, we evaluated both the performance of the algorithm and the accuracy of the state-dependent fault localization process through a set of experiments.

[1] http://www.cs.waikato.ac.nz/ml/weka/

1.4 Thesis Organization

Chapter 2 provides the background necessary to understand the contributions of this thesis, in addition to summarizing surveys of related work with regards to software fault localization techniques, data mining and software fault models. Chapter 3 provides the UML-based taxonomy of software faults that we created to understand the nature of different faults that may occur in object-oriented software. Chapter 4 illustrates our exploratory work using Weka classifiers on the concordance subject program, and provides details on the results published in ASE2009 [4]. In addition, it extends that work and applies it to new Java subject programs that we used to evaluate the new proposed technique. Chapter 5 presents the proposed state-dependent fault localization process and the customized mining algorithm "Associated sequence mining", in addition to the experimental design and results for evaluating the proposals and claims made in the thesis. Finally, chapter 6 provides the conclusion and future work.

Chapter 2

Background and Related Work

The background of this thesis is built upon finding the answers for two main questions. What are the existing fault localization techniques? What are the existing data mining techniques and how can they be used for fault localization? Sections 2.1 and 2.2 provide a summary of a survey done to find answers for these questions. During the exploratory stage of the thesis another question came up and that was, what is it exactly that we are looking for while performing fault localization? Therefore a survey of different software fault models and taxonomies was performed, and is summarized in section 2.4.

2.1 Fault Localization

Figure 2.1 shows a taxonomy of software fault localization techniques based on their underlying approach. As described by Ducasse [21], software fault localization techniques can be divided into three categories, namely Verification, Checking and Filtering. In verification, the source code is compared to a formal specification of the program and the mismatching parts are considered erroneous. In contrast to formal specification, checking relies on comparing the source code to knowledge about the programming language. Finally, filtering aims at pruning parts of the code that could not participate in the failure. More details about each of these categories, their subcategories and some examples are provided below.

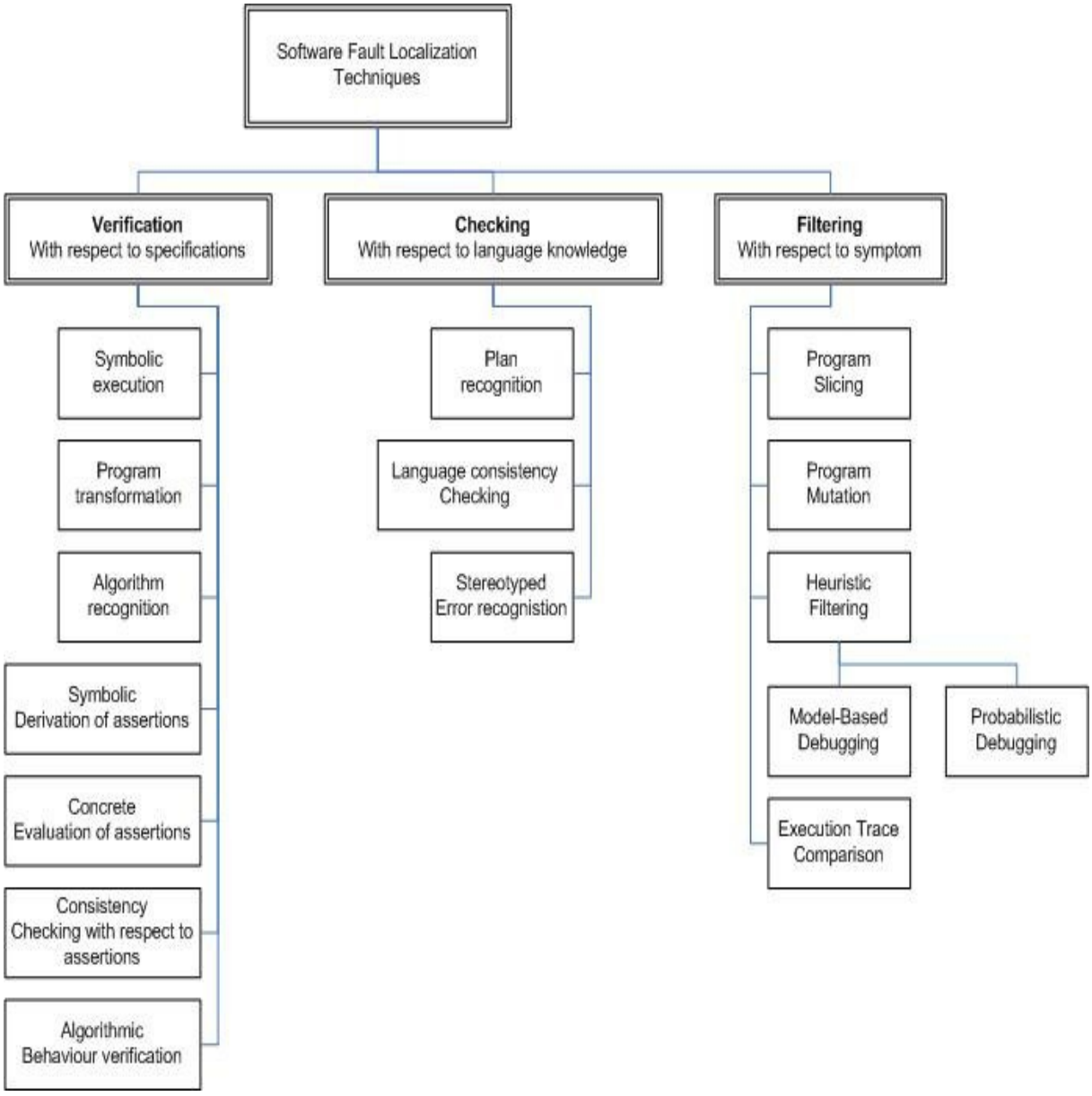

Figure 2.1: Taxonomy of software fault localization techniques

2.1.1 Verification

In general, the verification mechanism starts by comparing the source code and formal specifications of each component in the software under study. If the source code matches the specifications, the component is marked correct; otherwise it is marked suspicious.

"Doc2Spec" proposed by Zhong et al. [52] is an example of a verification technique in

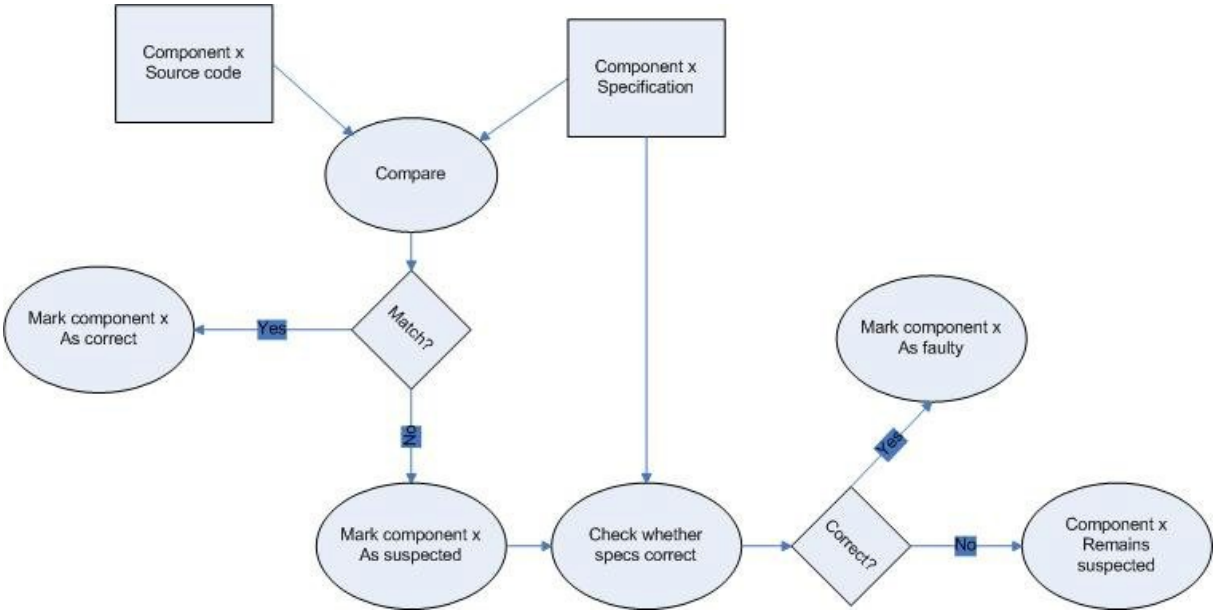

Figure 2.2: Verification

which Java APIs are detected automatically from the documentation, then used to locate faults in client programs using those APIs while violating the specifications.

Before a final judgement is given to the component, the specification itself has to be checked to ensure that it is correct, hence the component is erroneous. And here resides the main disadvantage of this mechanism. Since a formal specification usually does not exist at all, and if it exists it might be completely different than the final program, and/or it could be erroneous in itself, the mechanism is only suitable for small (toy) programs, or as a step in a localization process that combines different techniques.

The general steps of the verification process are shown in figure 2.2. The verification category can be divided into seven subcategories, namely, symbolic execution, program transformation, algorithm recognition, symbolic derivation of assertions, concrete evaluation of assertion, consistency checking with respect to assertions, and algorithmic behavior verification. A brief description of each of these subcategories is mentioned below; more details are provided by Ducasse and Emde [20].

In *symbolic execution* the program is represented by a model program, and a theorem prover

is used to formally verify the equivalence of the model program and the actual program. For example, Artzi et al. [6] used symbolic execution in combination with other techniques to localize faults in PHP applications. In contrast, in *program transformation*, the model program and the actual program are changed in parallel until either they match or diverge.

For *algorithm recognition* the specifications of the intended program are represented by a set of algorithms. The implemented program then is verified against those algorithms to locate the faulty parts. This is as opposed to *symbolic derivation of assertions*, where the specifications are represented by symbolic assertions of program components. The specified assertions are compared to corresponding symbolically derived assertions for the same component to specify whether the implementation of that component is suspicious or not. The *concrete evaluation of assertion* only differs from the preceding in that assertions are derived concretely as opposed to symbolically.

In *consistency checking with respect to assertions*, assertions that are used to specify properties such as types, mode or dependencies are compared to the actual properties of the program and highlight the component as suspicious if they are not consistent. Finally, *algorithmic behavior verification* assumes that the formal specifications of the whole program are present, and the user having these specifications can act as an oracle to the verification algorithm to compare the output of each computational step to its corresponding output in the specification.

2.1.2 Checking

The checking mechanism is similar to the verification mechanism in that it compares program components to previous knowledge; however, the knowledge used here is specific to the programming language as opposed to the functional formal specifications. This knowledge could be represented as known erroneous patterns, i.e. stereotyped errors, assumed programming conventions, or anticipated behaviour of a specific program component. Subcategories of the checking mechanism include plan recognition, language consistency checking and stereotyped error recognition.

In *plan recognition*, a plan is a stereotypic method of implementing a goal. In this type of fault localization technique, the system anticipates what the programmer is trying to do and compares it to the plan to see what went wrong. An example is what was proposed by Johnson and Soloway [30], a system called "PROUST". PROUST takes a Pascal program and a non-algorithmic description of what this program should do as inputs, and then it constructs a mapping between the two inputs using a knowledge base of programming plans and strategies along with the bugs associated with them.

For the *language consistency checking* subcategory, the program is supposed to adhere to well-formedness patterns or programming conventions. Program components are checked against those patterns. If the code violates those patterns it is highlighted as suspicious. An example of this approach was presented by Thummalapenta and Xie [47], who proposed a system called "Alattin" that mines alternative acceptable patterns of conditions surrounding Java APIs from code examples to reduce neglected conditions.

On the other hand, the approach of *stereotyped error recognition* is the opposite of language consistency checking. The code is searched for known erroneous patterns which are highlighted for further investigation. Livshits and Zimmermann [35] proposed a system called DynaMine that combines mining (i.e. analyzing) revision history and dynamic checking. The first is used to extract error patterns from historical records, and if any of them was discovered in the program under study, the discovered patterns are passed to dynamic checking to verify whether they are actually erroneous. Another use of data mining techniques for the same purpose is presented by Lo et al. [36], who use feature selection to extract the program behavior features that distinguish the faulty patterns. These features are then used in a classification algorithm that learns which values associated with these features actually represent errors by analyzing previous runs. The classification model is then used with new programs to locate program parts that can be classified as faulty.

2.1.3 Filtering

In filtering the focus is to filter out parts of the code that do not relate to a specific error symptom (i.e. failure), leaving the rest of the code under suspicion. Note that the objective is not to give final judgment of the identified parts of the code, but rather point them out for further investigation by the programmer or other techniques.

Sub-categories under filtering could be program slicing, program mutation, heuristic filtering and execution trace comparison. These subcategories are described below.

The idea of the *program slicing* approach is to compute the statements that contribute to calculating the value of a variable (or set of variables) at a specific point in the program. For example, a slice that satisfies the criteria $< 10, x >$ contains all statements necessary to compute the value of variable x at line 10 [49].

Chen and Cheung [14] illustrate the idea of program dicing, which is an additional analysis step for the program slices. It narrows down the suspicious lines by comparing the statements of a correct, or apparently correct, variable slice to those of an incorrect variable slice and highlights the differences as most suspicious.

In the *execution trace comparison* approach, execution traces of successful and failing test runs are compared; the differences between them are highlighted as suspicious. Many of the famous localization techniques belong to this category, including Tarantula [31], where the suspiciousness of each statement is calculated by comparing the number of failing test cases that execute this statement to the number of successful ones that executed it, according to the formula

$$\frac{\%P(s)}{\%P(s) + \%F(s)} \tag{2.1}$$

where % P(s) is the percentage of passing test cases that executed the statement "s" and % F(s) is the percentage of failing test cases that executed the statement "s".

Abreu et al. [2] found that using Ochiai 2.2 as a suspiciousness measure lead to slightly

better results.

$$\frac{f(s)}{\sqrt{tf * (f(s) + p(s)}} \tag{2.2}$$

where f(s) is the number of failing test cases that executed statement s, p(s) is the number of passing test cases that executed s, and tf is the total number of failing test cases.

Zeller [51] proposed a technique to isolate cause-effect chains in a program by comparing the values taken by each variable in a successful run by those taken in a failing one. Dallmeier et al. [17] also proposed a comparison-based approach that compares the sequences of calls that are more probably related to the failure. Renieris and Reiss [46] addressed the issue that in order for the comparison approach to be successful, the compared failing and successful runs must be quite similar; hence they proposed to choose the runs to be compared based on a similarity measure, that is, nearest neighbor queries. Cheng et al. [15] proposed comparing failing and successful runs after they are transformed into graphs, which are analyzed to find the most discriminative subgraph using the graph mining algorithm LEAP.

In *program mutation*, small parts of the program are changed to see if the change will cause the failure to be eliminated; if the failure still exists then the changed component is not considered suspicious. Papadakis and Traon [44] proposed a technique of this category, in which they calculate the suspiciousness of different mutants using the Ochiai [2] suspiciousness measure to identify the components that are more related to the failure.

Finally, *heuristic filtering* involves having an a-priori hypothesis about the correctness of some parts of the code. This can be feasible only if the hypothesis can be back-tracked. The a-priori hypothesis can be statistical, as in the probabilistic debugging where potential faults are calculated, and then the statements most likely to contain the errors are determined by a belief network (e.g. Bayesian Network) [11].

Model based software debugging is an example of the heuristic filtering approach. Model based software debugging is the application of model-based diagnosis (MBD) used to diagnose faulty components in hardware systems. In MBD the system description, i.e. the model, is used

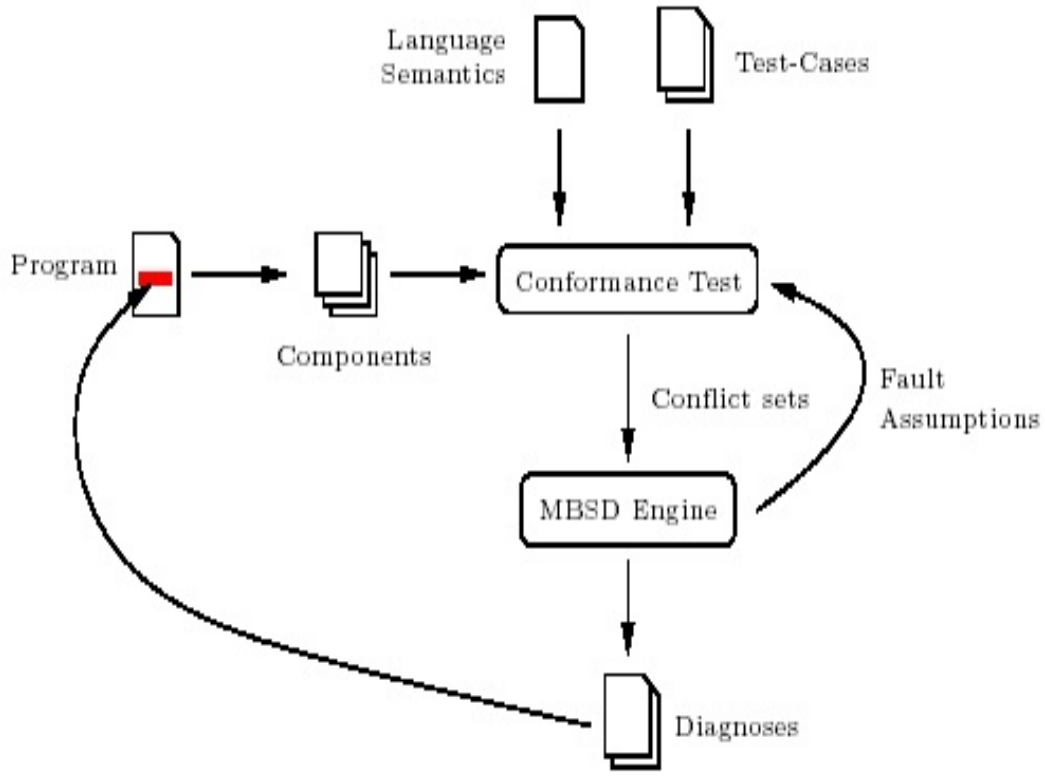

Figure 2.3: Model Based Software Debugging

to derive the correct behavior of the system; this behavior is then compared to the observed behavior of the physical system. The discrepancies between the derived behavior and the observed behavior are then used to find the diagnosis [49].

Since it is almost impossible to find a model of the correct behavior of the program, the model here represents the actual program (containing the faults), while an oracle or test case specifications are used to compute discrepancies between intended and actual results [49].

As Figure 2.3 illustrates, the first step in model based software debugging is to construct the model representing the program; the model consists of the components of the program along with relationships between them. The conformance test fails when the behavior of the program conflicts with the desired results of the given test cases. Then, using language semantics, the

components responsible for the conflict are computed and sent to the MBSD engine to compute fault assumptions (i.e. variations of program expressions or components) which re-sends them to the conformance test to be examined. At the end, the assumptions responsible for the conflict are translated into program locations and highlighted in the source code [39].

2.2 Data Mining

In this section, we provide background about data mining, including its definition as well as the different categories of tasks that can be performed using data mining techniques. In addition to that, we review existing fault localization techniques that utilize these tasks.

2.2.1 Definition

As defined by Hand et. al [27], data mining is the analysis of large observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner; these relationships and summaries are referred to as models or patterns.

The definition implies that there are three main components: a dataset to be mined, some desired results and a learning process for the transformation of the given dataset to the desired results.

2.2.2 The dataset to be mined

The representation of the data

The data to be mined could be stored and represented as structured data (traditional), like most business databases, semi-structured data, like electronic images of business documents and medical reports, or unstructured data, like visual and multimedia recording of events. The standard model of the structured data is a collection of cases (“samples” or “rows”); measure-

ments (“features”, “columns”, “variables” or “dimensions”) are measured over the cases [33].

The size of the dataset

One of the factors that motivated data mining is the great amounts of data stored in databases; these amounts of data prevented the data analysts from using them efficiently [25]. Also, large datasets are supposed to give more valuable information since data mining can be considered as a search through a space of possibilities; large datasets provide more possibilities to evaluate [33]. Therefore data mining techniques were designed to deal with large amounts of data, which means that applying data mining techniques on small amounts of data may yield inaccurate results [25].

On the other hand, too large data, especially when complex analysis is needed, will cause the analysis to be very time consuming, making such analysis impractical and infeasible. In this case, a reduced representation of the data is obtained through applying one of the data reduction techniques, which still maintains the integrity of the data while having a smaller size that needs less time to be analyzed [25].

2.2.3 The structure of the results

As mentioned by Hand et al. [27], the types of the structures produced by data mining techniques can be characterized by the distinction between a global model and a local pattern. A model structure is a global summary of a dataset which makes statements on each point in the measurement space; an example of such a model is the classifier produced by the classification algorithm (see classification). In contrast, the pattern structure makes statements only about the restricted regions of the space, an association rule (see dependency detection). By comparing the local pattern to the global model, outlying records can be discovered.

2.2.4 The data mining process

The main steps of the data mining process are: stating the problem, collecting the target data, preprocessing and/or transforming the data, mining the data (applying the data mining technique), then finally interpreting the results [33], [27]. A general description of each of these steps is given below.

Stating the problem

Since data analysis is usually associated with a certain application domain, domain-specific knowledge and experience are usually necessary in order to come up with a meaningful problem statement. For example, in dependency detection the data analyst may specify a set of variables for the unknown dependency and, if possible, a general form of this dependency as an initial hypothesis. This step requires the combined expertise of the application domain and the data mining task. In successful data mining applications, this cooperation continues during the entire data mining process [33].

Collecting the target data

Kantardzic [33] mentioned that there are two possible approaches to collecting the data to be mined. One of them is called "Designed Experiment", in which the collection of data is under the control of an expert (modeler), which results in experimental data which are collected exclusively for analysis purposes [27]. The other approach, which is the most commonly used, is called "the observational approach", in which the expert cannot influence the process of data generation, which results in "Observational data" that are collected for some other purposes rather than data mining analysis [27]. Using observational data eliminates the effort of applying restricted techniques for data gathering and collecting by making use of the already existing data. On the other hand, observational data may contain missing values, noisy data or inconsistent data, which means that data needs to be cleaned; also it is necessary to transform the data into forms that are appropriate for mining [25].

Preprocessing and/or transforming the data

Generally data preprocessing may include outlier detection and removal. Outliers are odd values in the dataset, and their existence can seriously affect the results of the data mining technique. However, removing them may result in losing important data if these outliers did not result from mistakes, which is the common case. The harmful effect may be avoided using robust techniques, that are least affected by the existence of outliers. In addition the preprocessing step may include variable scaling; for example, one feature with the range [0, 1] and the other with the range [-100, 1000] will not have the same weights in the applied technique; they will also influence the final data-mining results differently. Therefore, it is recommended to scale them and bring both features to the same weight for further analysis. Also, methods for encoding and selecting features usually achieve dimensionality reduction. In other words, they provide a smaller number of informative features for subsequent data modeling [33].

Mining the data

Here one of the data mining tasks is performed. In general a data mining task lies in one of four main categories: Classification, Clustering, Dependency detection and Outlier/Deviation detection. More details on these categories are provided below.

Interpreting the results

Usually the results of data mining applications are used in decision making. Hence such results should be interpreted in a form that is understandable by the user of these results [33]. This interpretation may be in the form of visualization.

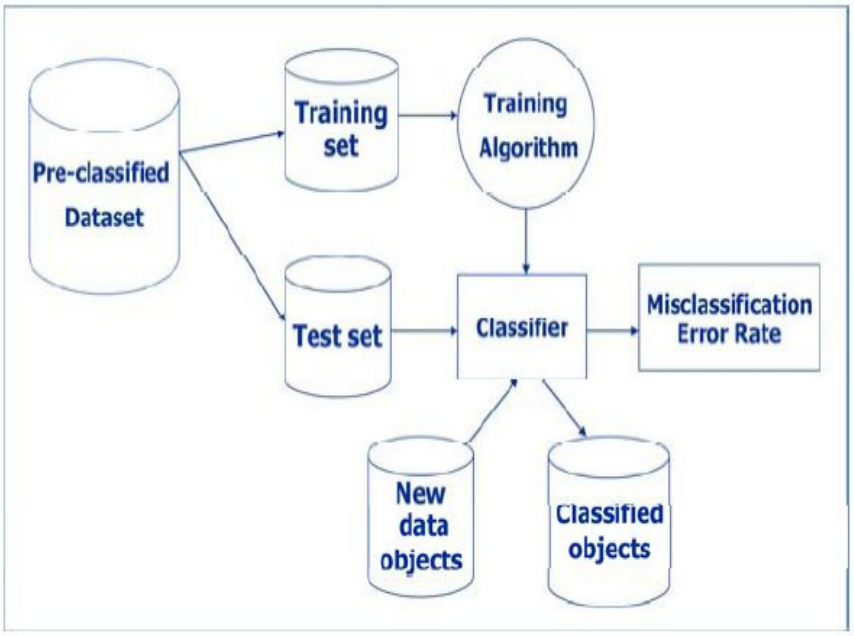

Figure 2.4: Taxonomy of software fault localization techniques

2.2.5 Main categories of data mining tasks

The data mining tasks can be categorized in four different categories: classification, clustering, association analysis and outlier detection. While the first three categories deal with knowledge that applies to the majority of the dataset, outlier/deviation detection tries to identify the odd (outlying) objects, as they are considered significant knowledge rather than errors or mistakes.

Classification

In classification the objects in the original data set are assigned to pre-specified classes (for example, a dataset of customers is classified into buyers and non-buyers); the goal of the data mining task is to understand the parameters affecting this assignment, and hence putting them in a model called "the classifier". The classifier can then be used to predict the class of a new unclassified data object (e.g. a potential customer) with a reasonable error rate [1].

In general, the classification task is performed as illustrated in Figure 2.4. As shown in the figure, the pre-classified dataset is divided into two subsets. One of them is used as a training

set, which is used by the training algorithm to generate the classifier, and the other subset is used as a test set to evaluate the classifier; if it gives a reasonable error rate it can be used to predict the class of new objects [1].

One of the techniques that can be used for classification is the decision tree, which can be defined as a predictive model that can help with making complex decisions. A decision tree may start by asking a question that has a series of answers; depending on these answers, further questions and answers may follow. Each branch of the tree is a classification.

Clustering

In contrast to classification, the goal of the clustering task is to classify an unclassified dataset. In clustering, groups of closely related data objects are identified; these groups may be further analyzed to determine the characteristics of the defined groups. For example, buying habits of multiple population segments might be compared, to determine which segments to target for a new sales campaign [1].

Clustering analysis is performed in two main steps. The first step is to measure the closeness of each pair of objects in the dataset; this closeness may be expressed in terms of distances between objects or in terms of similarities between them. The second step is to use some clustering technique to identify the clusters based on the values generated in the first step.

Association Analysis

The goal of the tasks in this category is to find correlations between different items. This correlation may be in the form of co-existence or co-occurrence; if this correlation is analyzed on the basis of a fixed point of time, then the analysis is called Association Analysis. A typical example of such analysis is the market basket analysis, where association rules are generated to help in designing effective marketing campaigns and to rearrange merchandise on shelves. To use an example that is commonly used in the literature, an association rule may indicate that those customers that purchase bread and butter also purchase milk [1]. More formally, an

association rule is a rule of the form

$$\text{R: } IF\{AttSet1\}THEN\{AttSet2\}$$

where AttSet1 is called the antecedent and the AttSet2 is called the consequent and

$$AttSet1 \bigcap AttSet2 = \emptyset$$

For example, a typical association rule may be as follows:

$$IF\{bread, butter\}THEN\{milk\}$$

Since a great number of rules may be produced during association analysis, there must be some way to measure the significance of each rule. Two measures may be used for this purpose. One of them is the *support*, which is the probability that a randomly chosen row (object) from the dataset contains AttSet1; the other measure is the *confidence*, which is the conditional probability that a randomly selected object of the dataset will contain AttSet2 given that it contains AttSet1. In terms of the support, the confidence can be calculated as shown in Equation 2.3.

Another measure that can be used to further evaluate a rule is the lift, which can be calculated as shown in Equation 2.4. Lift would indicate that the occurrence of the antecedent increases the probability of the occurrence of the consequent in the same transaction if its value is greater than 1, and indicates the opposite relationship if its value is less than 1. A value of 1 would indicate that there is no correlation between the two parts of the rule.

$$confidence(R) = \frac{support(AttSet1 \bigcup AttSet2)}{support(AttSet1)} \tag{2.3}$$

$$lift(R) = \frac{confidence(R)}{support(AttSet2)} \tag{2.4}$$

Another type of dependency detection is sequence-based analysis, where the co-existence of items is analyzed on a time-interval basis, and other factors such as the order in which those items appears and the amount of time between their occurrence are also taken into account [1].

Outlier/Deviation detection

Outliers are generally the odd objects in the dataset. Outliers can be viewed from two different perspectives: one of them considers the outliers as undesirable objects that should be treated or deleted in the data preparation step of the data mining process, and the other considers the outliers as interesting objects that are identified for their own interest in the data mining step of the data mining process. In the latter case outliers should not be removed in the preprocessing step, which is why one of the main categories of tasks performed by data mining techniques is the outliers/deviation detection. Applications that make use of such detection include credit card fraud detection in e-commerce applications, and network intrusion detection in the field of network security [8].

2.3 Data Mining Applied to Fault Localization

The taxonomy of fault localization techniques shown in figure 2.1 classifies the different techniques based on the underlying approach. Each of these approaches can be implemented using different analysis techniques. Some rely on direct statistical analysis; others rely on more complicated analysis like machine learning and data mining.

We are particularly interested in the use of data mining for fault localization. Initially we were inspired by the work done by Denmat et al.[18] that suggested that Tarantula [31] can be seen as an association analysis task. Tarantula as a fault localization technique is searching for the single statements that usually cause a failure in the program when executed; this can be reformatted into the association rule:

$$IF\{e(s)\}THEN\{F\} \tag{2.5}$$

That is, if a statement s is executed then probably the program fails. Tarantula's suspiciousness measure can be transformed into a formula based on lift, see Equation 2.4, as shown in equation 2.6.

$$Suspiciousness(s) = \frac{lift(IF\{e(s)\}THEN\{F\})}{lift(IF\{e(s)\}THEN\{S\}) + lift(IF\{e(s)\}THEN\{F\})} \quad (2.6)$$

Cellier et al. [13] extended this notion to finding association rules that contain multiple lines in the antecedant of the rule, then organizing these rules into a concept lattice in a way that helps the programmer/debugger to identify the parts of the antecedents that are more common among the different rules. This type of analysis still suffers from the same problem that Tarantula suffers from, which is mainly highlighting disjoint code lines that are scattered throughout the program, instead of highlighting higher constructs as the programmers would prefer.

Yu et al. [50] also used the task of association rule mining for fault localization; however, their approach is to find the sequence of graphical user interface (GUI) events that led to a failure, hence considering the event handler that most commonly called by those events as the suspicious part of the code. Under the same category of execution trace comparison, another mining technique is used by Cheng et al. [15] after transforming the traces into graphs and comparing them using the graph mining algorithm LEAP. LEAP can be considered a variation of association mining for graph data.

Furthermore, data mining can also be used in checking techniques. For example, "Alattin", a language consistency checking technique, also uses association mining to find acceptable patterns of conditions surrounding Java APIs [47]. DynaMine [35] is another example of using association mining in fault localization, which mines historical records for error patterns to be used as stereotypes to check new code. Lo et al. [36] and Murtaza et al. [40] also proposed using data mining for stereotyped error recognition; however, they utilized the classification task.

2.4 Fault Models

It is crucial for software development in general and fault localization in particular to understand the nature of the different problems that can occur in software. In this section we try

to understand the nature of those problems starting by providing generic definitions in section 2.4.1 . The generic definitions help to provide scope to our research; however, they are not sufficient to understand the nature of what we are looking for in fault localization, that is, the different types of faults. Fault models provide a structure of different types of faults based on certain criteria. In sections 2.4.2 through 2.4.5 we summarize the fault models provided in the literature.

2.4.1 Definitions

Wieland [49] has mentioned definitions of some terms that are used to refer to problems in software. In order to specify our focus we mention those terms and their definitions below:

Mistake: Human action that produces an incorrect result. For example copy and paste error, spelling mistake, etc.

Failure: An incorrect result. Other sources refer to the same concept as symptom.

Error: The difference between a computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition.

Fault: An incorrect step, process or data definition. This concept is what we are focusing on. Some sources [16] refer to it as "defect" and others refer to it as "anomaly" [41].

2.4.2 Orthogonal Defect Classification

Orthogonal Defect Classification (ODC) was developed by IBM [16] with the objective of providing a link between the cause of the defect, represented by the associated activities of the different stages of development, and the effect, represented in the change in the shape of the reliability growth curve associated to a specific defect type.

As the name suggests, ODC aims at grouping software defects into classes that can be mapped to a specific activity or stage of the development process, more like a point in a Cartesian system of orthogonal axes by its coordinates (x,y,z). As we can see in Figure 2.5, ODC

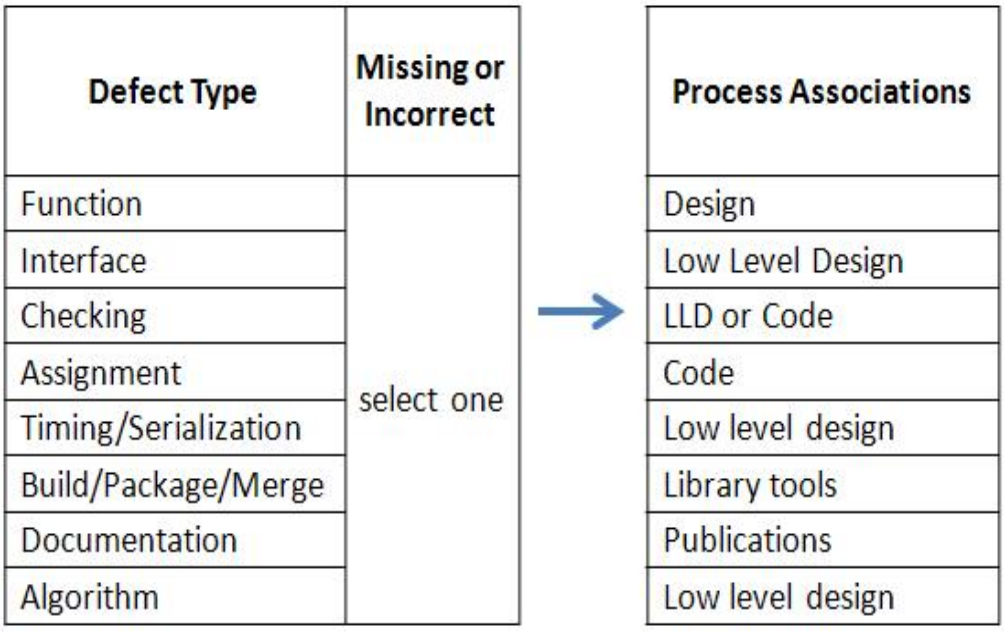

Figure 2.5: Orthogonal defect classification

contains eight different classes of defects; each of them could be viewed as a defect either because it is missing or because it is incorrect.

Each of the classes can be directly mapped to the stage of the process that needs the attention in order to fix the defect. The definition of each of these classes is available in Chillarege et al. [16].

2.4.3 Comprehensive multi-dimensional taxonomy for software faults

Inspired by the orthogonal feature of IBM's ODC, different software fault taxonomies can be combined in one comprehensive multi-dimensional taxonomy combining different angles of viewing a software fault in one taxonomy.

Figure 2.6 represents the suggestion of a five-dimensional space for classifying software faults. We now look at each of these dimensions in more detail.

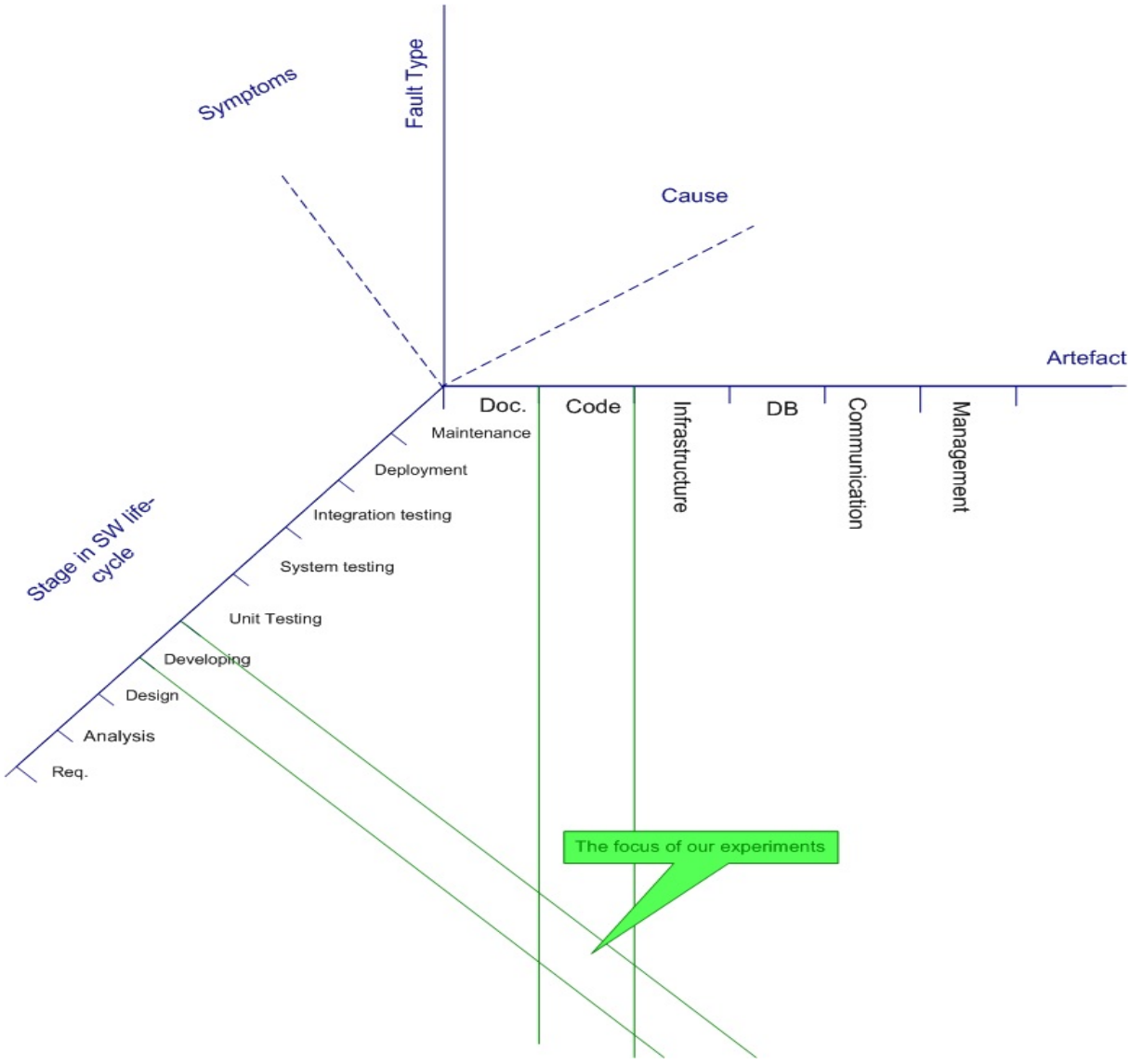

Figure 2.6: Multidimensional classification

Stage in the software life-cycle dimension

This dimension might seem like "Project phase - RR200" category of IEEE's standard classification for software anomalies [41] recognition classification scheme; however, it is different in that this dimension represents the stage where the faults belong, while IEEE's standard refers to the stage where the defect/anomaly was discovered or recognized.

This is also different than the associated stage of the ODC [16] in that ODC's stages refer

to the stage where the defect can be fixed, which is not necessarily the same stage where the actual defect belongs, e.g. developer might find a detour to work around the problem instead of directly fixing it.

The work done on Hayes' verification and validation research project to build a requirements fault taxonomy for NASA [29] belongs to the requirements part of the stages dimension.

Software artefact dimension

The artefact dimension represents the part of the software that contains the fault; including documentation, code, underlying infrastructure (e.g. operation system), the database, communication (e.g. network topology, data transmission, etc.) and the management (e.g.plans, procedures). This is similar to IEEE's standard classification for software anomalies [41] "Source - IV200" criteria of classification in the investigation step. However, the same level of granularity of IEEE's standard can be achieved by specifying the corresponding value of the stage of the software life-cycle dimension. For example, the requirements (IV211) subclass of the specification class (IV210) corresponds to the documentation on the "artefact" dimension and requirements on the "stages" dimension.

Similarly, the data subclass (IV217) of the specification (IV210) class corresponds to database on the "artefact" dimension and the requirement on the "stages" dimension.

Fault-Type dimension

ODC [16] provides eight classes of defects. Some of them refer to defects in the code, e.g. assignment error and interface. Others refer to defects in the deployment, e.g. build/package/merge. Finally some of them refer to defects on the functionality of the program e.g. function.

IEEE standard 1044 [41] provided eight subtypes of the type category (IV300) in the investigation classification scheme. Most of them refer to errors in the code, some refer to errors in the data, and others refer to errors in the documentation.

The work done by Mariani [38] and Bruning et al. [10] represent a different view of classi-

fying software faults focusing on faults specific to certain software architecture or feature. For example Mariani [38] focused on the inter-component connectivity of the component-based software while Bruning et al. [10] classified faults based on the steps of the SOA process.

Another view is based on the programming paradigm. For example, Hayes [28] provides a taxonomy for errors specific to OOP.

Cause dimension

The term cause was used in both the ODC [16], referring to the stage of software development that needs the attention to fix the fault, and IEEE's standard [41], referring to the reason that lead to the problem. The latter view is closer to our suggestion here; however, the entries that represent the values of that dimension are quite different than classes and subclasses mentioned in IEEE's classification. Most of the latter were captured in the artefact dimension.

Values that can be taken on that dimension include human limitations (e.g. mis-typing), system upgrade, bug fix (e.g. sometimes fixing a defect may cause other areas in the code to be defective) and hardware failure.

Symptoms dimension

This dimension represents the different types of observable failures. It is similar to the symptom category (RR500) of the recognition classification scheme of IEEE Standard [41]. Class A of defects related to the understanding of the problem, as mentioned in Endres' Classification of Software Defects [12], might fit in this dimension.

The suggested multi-dimensional taxonomy can be extended by adding other dimensions like severity (e.g. Moderate, serious, disturbing) [9] and effect on the reliability growth curve.

2.4.4 How can this multi-dimensional taxonomy be useful?

This taxonomy could be useful for researchers to determine the scope of their research; e.g. in Figure 2.6 the scope of our experiment is to find software faults that occurred in the develop-

ment stage and were found during system testing.

Also it could be useful to compare different testing and fault localization techniques in terms of the faults they can detect or discover; hence it can realise the gaps or the areas that are not covered by research.

In addition to that, this taxonomy can be implemented as a data warehouse to collect all information about the different types of faults, and hence be used to facilitate the debugging process. For example, when certain symptoms are observed, this data warehouse can be used to find out which fault types might be associated with those symptoms and which artefact might contain these faults.

2.4.5 UML-based fault models

Trong et al. [19] proposed a fault taxonomy for UML design models, for the objective of being used in verification and validation of design models. The first level of the taxonomy has three categories. One of them is for the design metrics related faults, like faults related to cohesion and coupling. Obviously, this category is far from being related to the focus of this thesis, as our focus is on the implemented code and faults related to it. The second is for faults detectable without execution, which is related to faults in the syntax of the diagrams either in a single diagram or in the consistency between different diagrams. Again, this category is not related to the focus of this thesis as we are interested in the faults that produce failures, i.e. run-time faults, while the faults in this category may correspond to compile time faults in the code. Finally, the third category is for the faults related to behavior. Some of the faults that belong to this category may correspond to implementation faults while others are merely design faults and would not have any effect on the behavior of the code and would not produce any failure. For example, a method that is given a public accessibility when it should be private in the definition of an operation is a design fault that if implemented may lead to a failure.

Ammar et al. [5] also proposed a fault model to be used for verification and validation of UML models at the specification level. However, the focus of their model was based on an

extension for UML for real-time embedded software. Kundu and Samanta in [34] mentioned a fault model based on UML activity diagrams with the purpose of generating test-cases that target these faults. However, the definitions of the faults were at a generic scope at the use case level. Finally, Offutt et al. in [42] proposed a list of nine faults that relate to specific complex situations that can occur due to the use of polymorphism in object oriented programming.

2.5 Conclusion

After careful review of the literature we found that there were a number of research gaps that we aimed at filling them by the research done in this thesis. These gaps are described below.

1. There is not enough understanding of the nature of the faults that the different localization techniques are supposed to find.

2. The existing fault localization techniques do not account for both the advantages and challenges provided by the object-oriented paradigm.

3. Even the most successful localization techniques were not utilized by programmers, as they do not match their expectations and mindset.

Chapter 3

UML-Based Fault Model

The different software fault models and taxonomies mentioned in chapter 2 were helpful. However, they did not provide enough answer to the question that we were looking for, that is, "What is it that we are looking for while performing software fault localization?" More specifically, we needed to know what types of fault can occur in object oriented programs. In this chapter we use UML models as the basis to enumerate what the possible fault types are in object oriented software, with the objective of choosing some of them in order to target with the localization techniques studied in this thesis.

3.1 UML-based software faults taxonomy

UML diagrams can be grouped into two main categories, structural diagrams and behavioral diagrams [43]. Structural diagrams aim towards modeling the static structure of the system, while the behavioral diagrams aim towards modeling its dynamic behavior. Accordingly, software faults can be structural faults and/or behavioral faults. These two categories represent the first level of our taxonomy.

The second and higher levels of the taxonomy represent faults that would appear on the different UML diagrams if reverse engineering was applied on the source code to produce those diagrams. This should not be confused with faults in the design, as the same diagrams

are used during the design stage of the software lifecycle as well.

UML describes seven different diagrams under each category. The ones we focus on are the class diagram and object diagram of the structural diagrams, and sequence diagram and activity diagram of the behavioral diagrams. These four diagrams capture the majority of features that can be found in object-oriented software; other diagrams capture similar features from different viewpoints.

3.1.1 Structural faults

The structural features of an object-oriented program can be shown in class diagrams and object diagrams. Figure 3.1 shows the classification of the structural faults.

Class structure faults

The class diagram shows the different classes of the system, their attributes and methods, and relationships between them. Relationships between classes could be in the form of association, aggregation, composition and inheritance. An interface is an abstract form of a class. Faults that might appear on the class diagram fall under a category we call class structure faults. A class structure fault can be either a class definition fault or an association fault. The difference between these two categories is that the latter is related to faults in the relationships between the different classes.

A class definition fault could be an attribute definition fault, a method definition fault or an inheritance fault. Even though inheritance is a relationship between classes, it is considered under the class definition category as it affects the internal structure of the inheriting class.

An attribute definition fault can be either a faulty data type or a faulty initialization value. A faulty data type can cause failures that are not easy to explain, especially with languages that provide automatic casting, e.g. truncating fractions to save a value in an integer type variable; these truncations can accumulate over the execution of the program, causing significant errors. Also, if an attribute is initialized by a faulty value, it might lead to accumulated faulty steps

that would lead to a failure at the end. A method definition fault could be manifested in a faulty return type or a faulty parameter list. A parameter list can be faulty by having a faulty number of parameters, a faulty data type or a faulty initialization value. Inheritance could be faulty in two different ways. The first is to have an overload fault and the second is to have a depth fault. For example, if we have an Employee class that is a parent of the Executive class and a CEO class that should inherit from the Executive class, but CEO incorrectly inherits from Employee directly, then we have a depth fault. If, in contrast, the Employee class defines a method "calcSalary" that provides general calculations for the employee's salary that should be overridden in the Executive class to provide specific calculations for this type of employee, if this overload is not provided in the Executive class, then it is considered a missing overload fault. Now if the CEO class is supposed to inherit and use the same implementation provided in the Executive class, but instead it contains its own implementation of the same method, then it is considered an extra overload fault.

An association fault could be a multiplicity fault where the maximum is exceeded or the minimum is not achieved. Alternatively, it could be an association at a faulty level, for example, when a class is substituted by its subclass in the association.

The association multiplicity faults mentioned here should not be confused with the example mentioned in chapter 2 as an example to distinguish between design faults and implementation faults. To be more specific, a coding multiplicity fault would be manifested for example when defining an array of items with above the maximum number of items or below the minimum number of objects that should participate in the association.

Lastly, an association fault can be an association property fault, that is, when there is a fault in implementing a property of the association. For example, if the *isSorted* property of an association should be true and the association is implemented using an unsorted set object instead of a list object, then there is an association property fault.

State faults

An object diagram shows the structure of specific objects, their object states, and actual instances of associations between those objects. In other words, the object diagram shows the state of the system at a specific point of time. Therefore the category of faults that we can define on an object diagram is called the state faults. A state fault can be either an object state fault, class state fault or system state fault. An object state fault occurs when there is a problem with the consistency of the values of the attributes of one specific object. If the inconsistency involves more than one object then it is considered a system state fault. If that inconsistency is in the association between different objects at a specific time, then it could be object multiplicity (if there is a problem with the number of associated objects), or an object type fault (if the associated object is of a compatible type with the supposed object). If the inconsistency is between the values of attributes of different objects, then it is considered a state correlation fault. In addition to that, we can consider the values of static variables to represent the state of the class; if there is a faulty value of a static attribute then it is considered a class state fault.

3.1.2 Behavioral faults

UML's behavior diagrams are used to model the functionality of the system from different angles and at different levels. For example, the sequence diagram models the interaction between objects in terms of sending messages and receiving responses, while the activity diagram models the behavior of the system with focus on decisions, parallelism and sequence of intermediate steps/activities aiming towards achieving certain functionality.

A behavioral fault can be either an interaction fault or a method implementation fault. As shown in Figure 3.6, the interaction fault category and its subcategories are inspired by the modeling artifacts of the sequence diagram. The four subcategories under interaction fault are message fault, object lifetime fault, sequence fault and state invariant fault; these are the faults related to how the different objects interact together through messaging to achieve certain functionality. Internal interactions between the methods of the same object are also included in

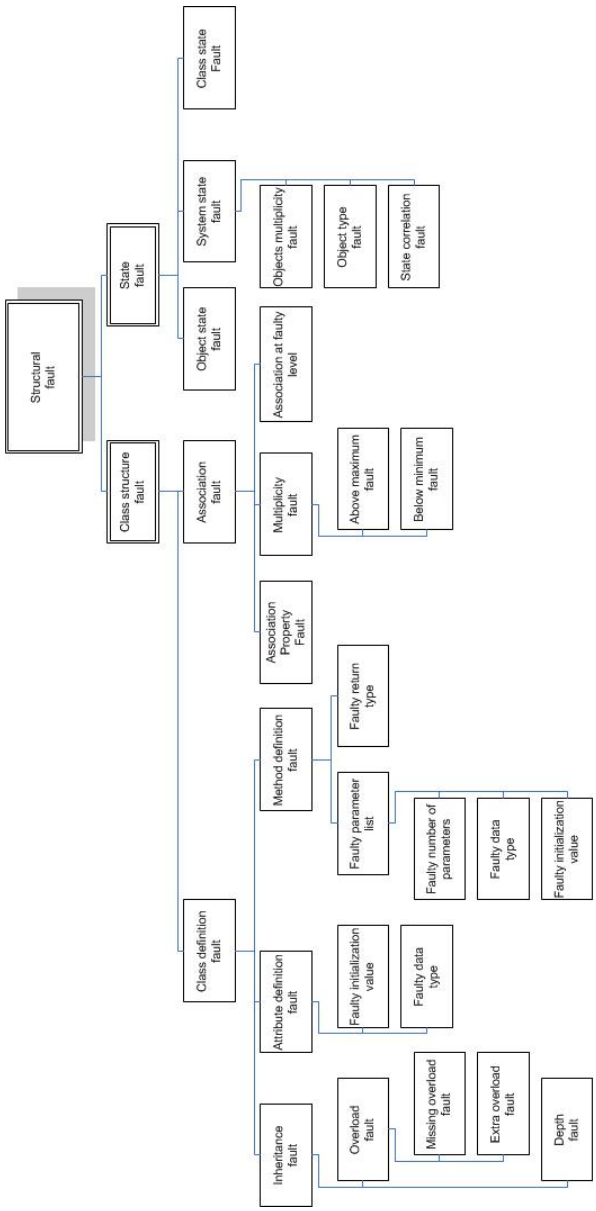

Figure 3.1: Structural Faults

this category. As shown in Figure 3.7, the method implementation fault category specifies the faults that might occur in the internal implementation of a method as inspired by the modeling artifacts of the activity diagram. More specifically we deal with statements that do not involve a message call as these statements are dealt with in the interaction category. The method implementation fault category has four subcategories: decision fault, forking fault, sequence fault and local statement fault. More details about the subcategories are provided below.

Interaction faults

Message faults, as the name suggests, are the faults related to the messages used for interaction. Typically, interacting through a message involves three components: calling the message through a reference to the object providing the service, sending values for the parameters required by the message, and receiving a return value as response to the message call. The faults related to each of these components constitute the subcategories of the message fault category.

A call of a message of the wrong object can happen if another object of the same type, an object of a parent type or an object of a child type is used instead of the intended object. In all these cases it might be hard to discover the fault since the called object might already contain a message of the same type but probably with different implementation or applying to different encapsulated data.

An argument fault can happen if the arguments are switched, faulty values are sent in the right positions or a faulty accessibility type is used, such as confusing a call by value to a call by reference or vice versa. Also some languages provide the ability to determine whether the parameter is a read only (IN parameter), write only (OUT parameter) or read-write (IN-OUT parameter); any mistake related to this feature falls under the category of faulty accessibility type.

A return value fault can be manifested as a faulty value sent by the function or as a value received by a wrong variable.

Figure 3.2 shows an example of a faulty object type in the fragment (b), and faulty return

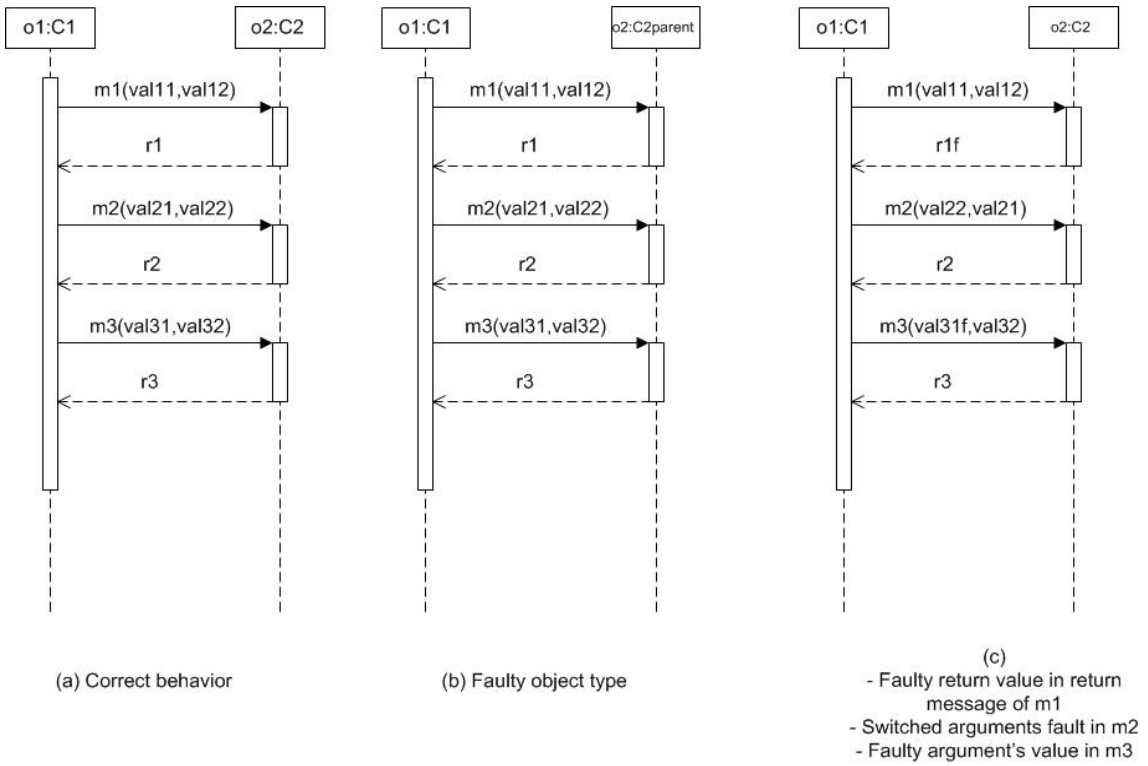

Figure 3.2: Interaction faults example

value in return message of m1, switched arguments fault in m2, and faulty arguments value in m3 in fragment (c).

The lifetime of an object can start by its construction and ends by its destruction. Hence, an object lifetime fault can be a construction fault or a destruction fault. The construction fault could be a call to a parent's or child's constructor, or in the case of using the same reference variable to refer to different objects, early or late construction faults. Similarly, a destruction fault can be manifested in an early or late destruction; for example, an early destruction can lead to using a null reference. Examples for construction and destruction faults are shown in figures 3.3 and 3.4 respectively.

The sequence diagram shows the message calls and returns in their chronological order from top to bottom on the objects' lifelines. The sequence fault category includes faults related to this aspect.

Along the lifeline of an object a message call could be missing. Similarly, the response of

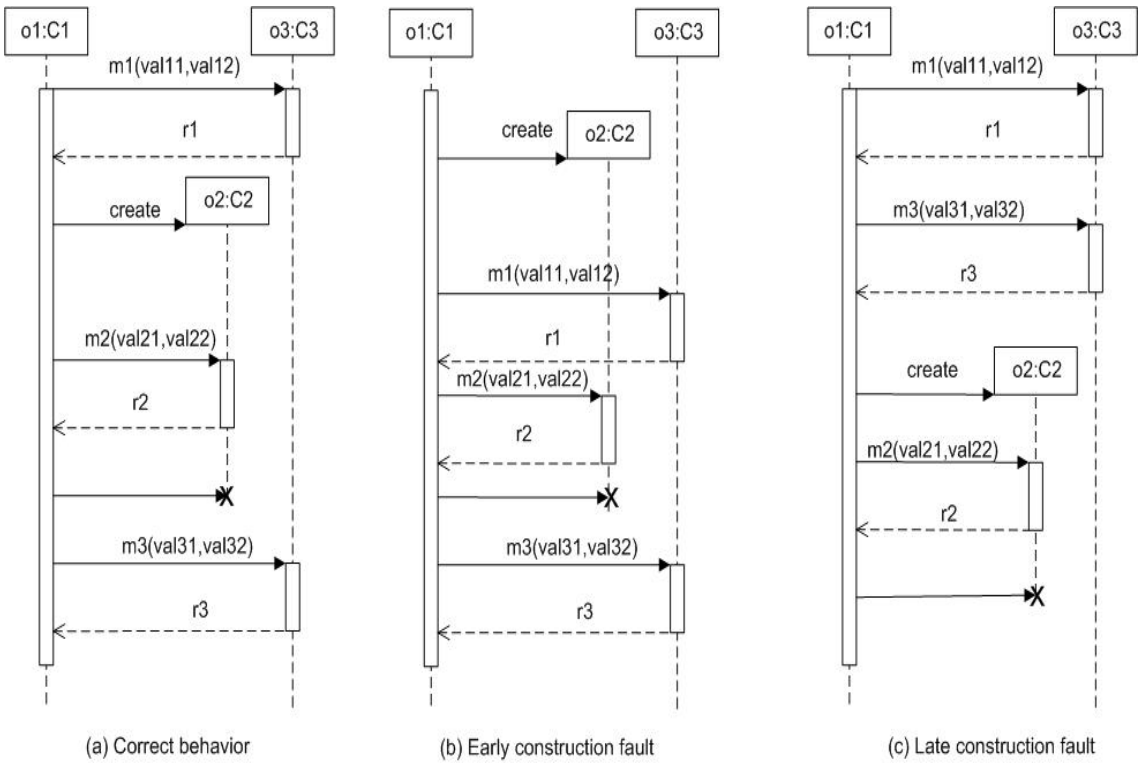

Figure 3.3: Construction faults examples

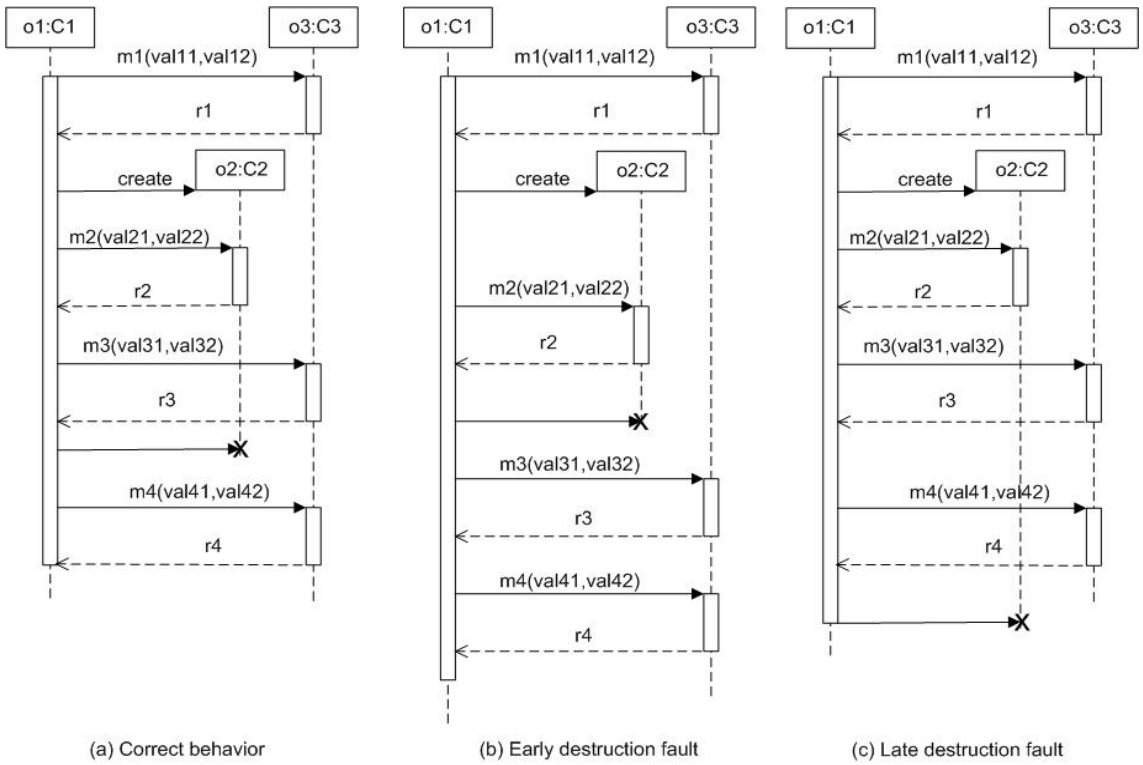

Figure 3.4: Destruction faults examples

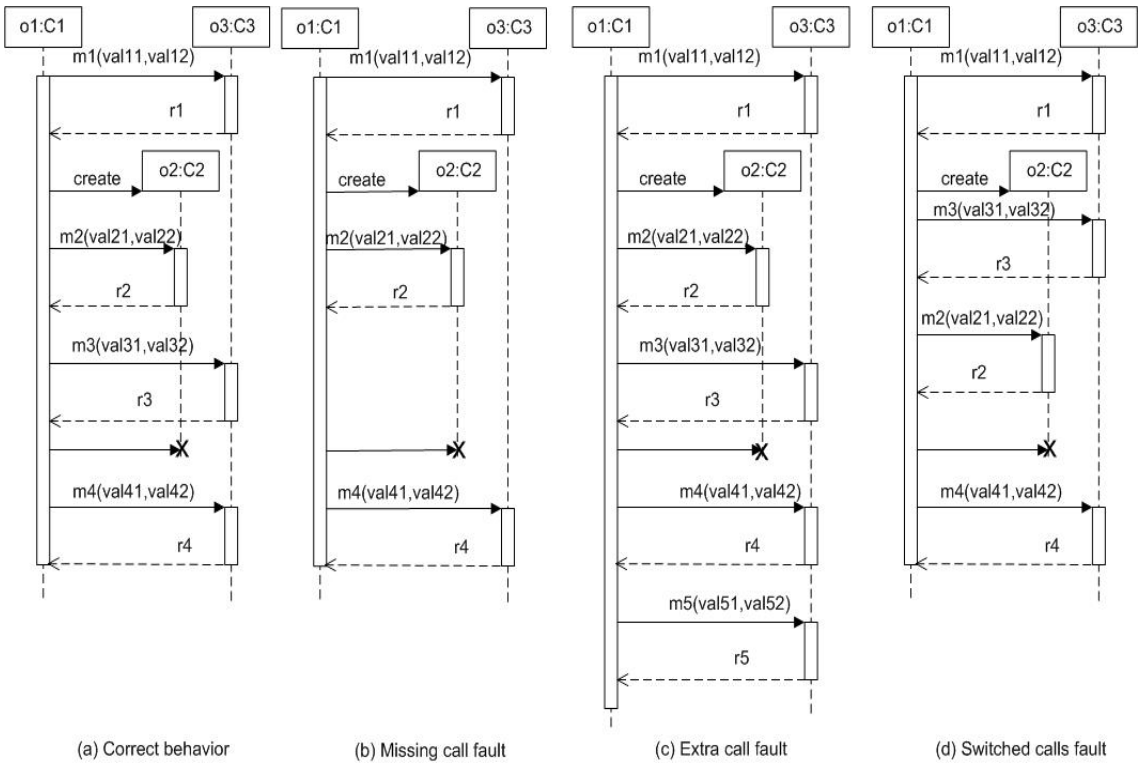

Figure 3.5: Sequence faults examples

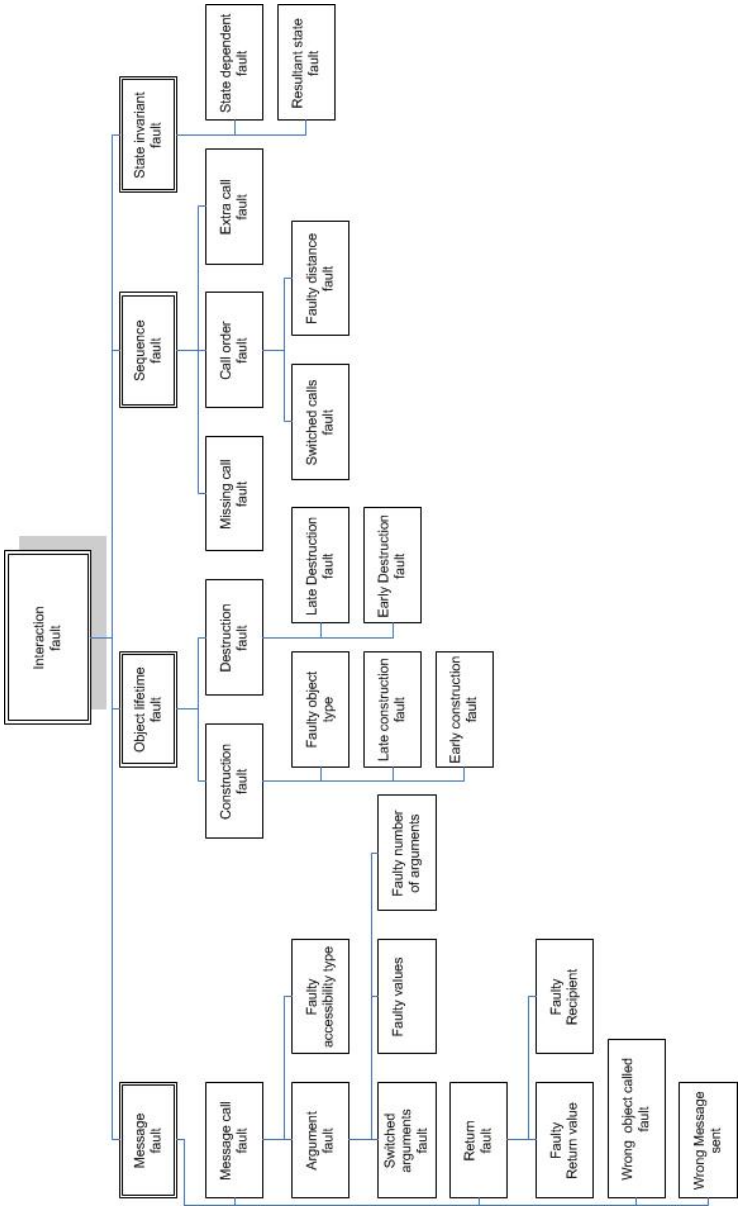

Figure 3.6: Interaction Faults

the call could be missing as well, for example, if the method returns an empty value or no value at all when it should return a value, or if the return value is not received in a memory variable and hence lost.

The order of the existing calls and returns can also be faulty in case two or more calls have switched places causing a switched calls fault or have an incorrect number of calls/returns in between, causing a faulty distance fault. Figure 3.5 shows examples of the sequence faults.

A state invariant is a construct that can be used on a sequence diagram to model the cases when the state of an object determines the way it responds to a message call [24]. The category of faults related to this construct is the state invariant fault category. If the state invariant is placed before an interaction, it represents the condition under which this interaction will take place. If the interaction is inconsistent with the specified state, then it is considered a state-dependant fault. On the other hand, if the state invariant is placed after the interaction, it represents the state that the object should be in after the completion of the interaction. If this state is not reached, then it is considered a resultant state fault.

Method implementation fault

Figure 3.7 shows the hierarchy of the types under the method implementation fault category. This category includes the types of faults that might occur in the internal implementation of a method if it would be modeled in an activity diagram. More specifically, this category deals with the statements that do not involve a message call as these statements are dealt with in the interaction category. The three subcategories under method implementation faults correspond to the faults related to the three main constituents of an activity diagram.

A decision fault could be a logical condition fault, for example a wrong comparison operator, branching fault, merging fault or branches fault.

A branching fault happens when a statement that should exist before the branching incorrectly exists after it in one of the branches (early branching fault), or when a statement that should be part of one of the branches actually precedes the branching (late branching fault).

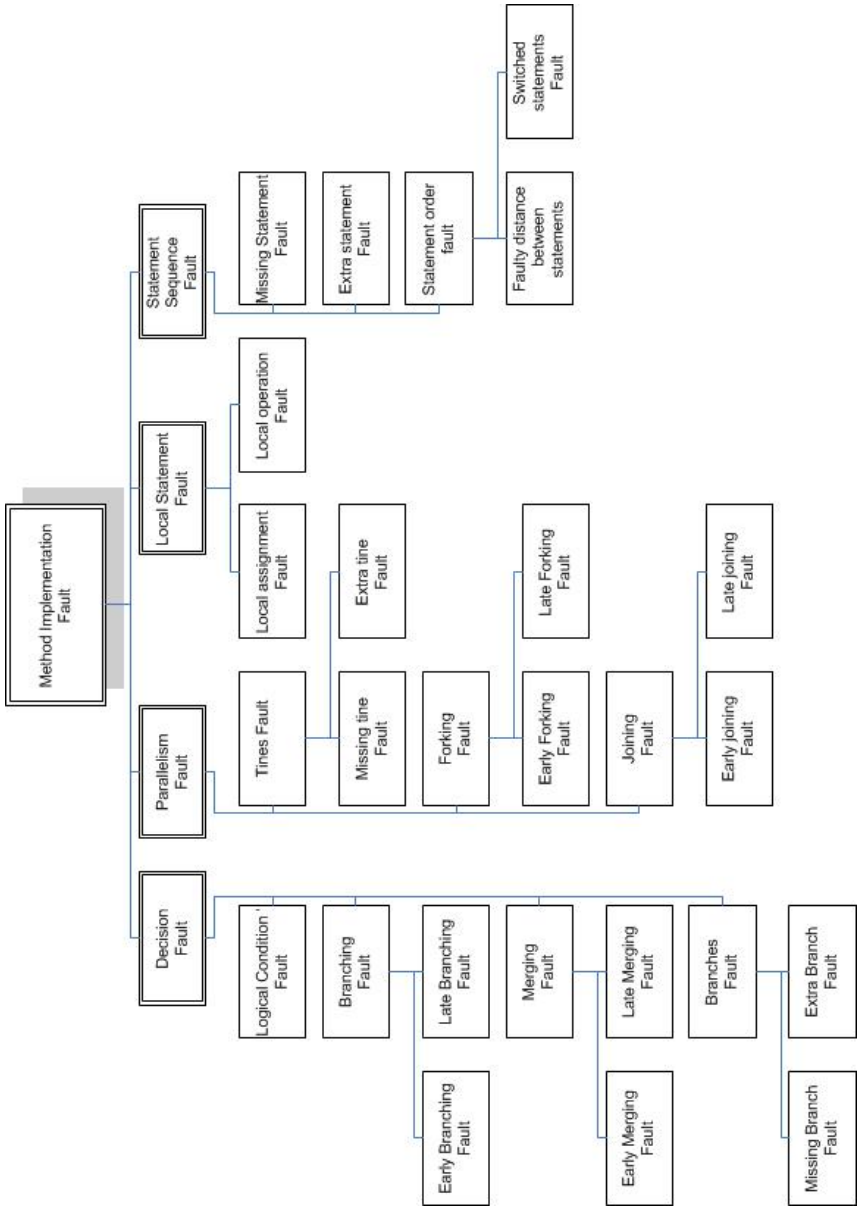

Figure 3.7: Method Implementation Faults

Similarly a merging fault happens when a statement that should exist after the merging of a branching statement, incorrectly exists before it as part of one of the branches (late merging fault), or vice versa (early merging fault).

A branches fault happens when there is a missing branch or an extra branch. The subcategories under the parallelism fault category are similar to those under the decision fault if we replaced the decision construct by a parallelism construct. A tines fault happens when there is a missing or an extra tine in the fork. A forking fault could be early or late forking and a joining fault could also be late or early joining.

Figure 3.8 shows an example of the correct behavior of a decision construct in fragment (a) and the correct behavior of a parallelism construct in fragment (b). Late and early branching, merging, forking and joining faults are shown in figures 3.12, 3.10, 3.9 and 3.11 respectively. Missing and extra branches and tines faults are shown in figures 3.13, 3.14 as well.

The statement sequence fault category is similar to the interaction-sequence category except that it deals with non-message-call statements.

Finally, a local statement fault occurs if a local assignment is faulty or a local operation is faulty.

3.2 Which faults should we focus on?

In reality, the defined types of faults are not strictly mutually exclusive; in addition to that, a fault yields a failure after propagating through a chain of infections [42] that can be considered as intermediate faults that can be of any type. For example, FAULT_101 of JMeter version 5 replaces the statement return getName() + getValue(); by return getName() + getMetaData() + getValue();. We classified that fault as a local operation fault, as it contains an extra concatenation operation; however, it can be also classified as an extra call fault, as the call to getMetaData() is extraneous. In both cases it will lead to a faulty return value, which will lead to another fault based on how the caller method would use that returned value. For example,

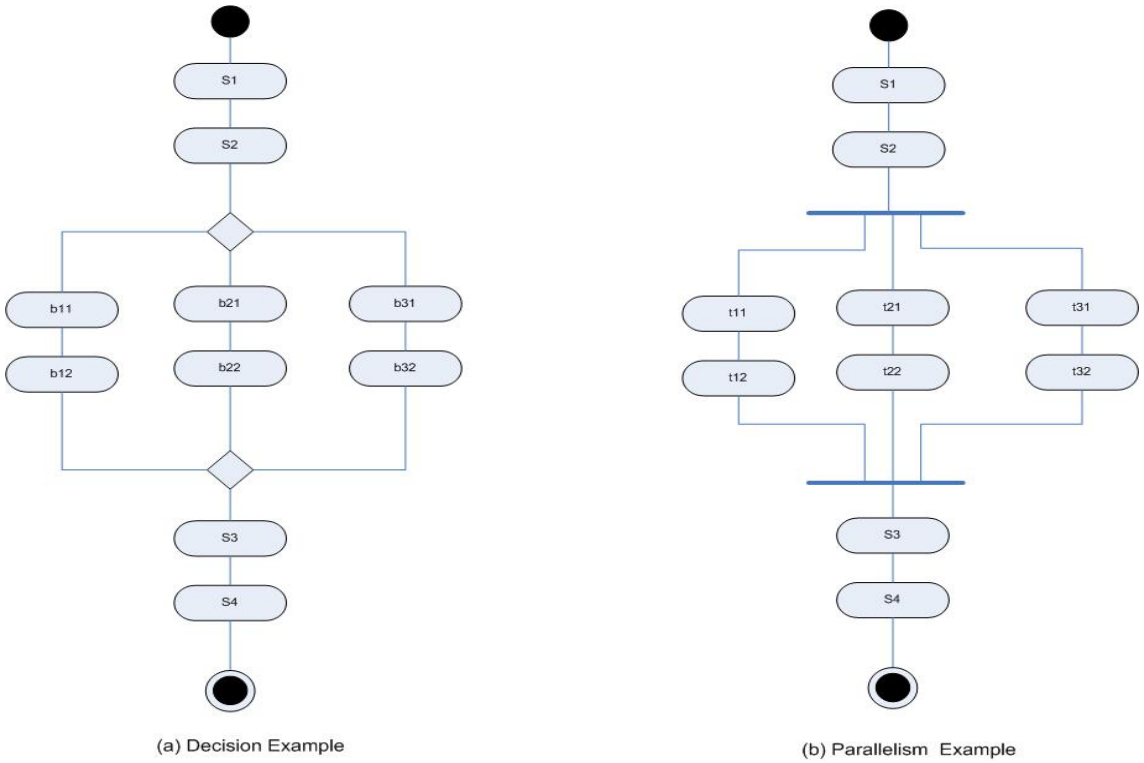

Figure 3.8: Example decision and parallelism constructs

if the caller uses the returned value in a condition of an if statement, that would be considered a logical condition fault, and if that if-statement somehow uses a value of an attribute of the object, then it will lead to a state-dependent fault and so on.

The different combinations of fault types in a propagation chain is enormous, which makes the deductive approach towards the decision of which type we should focus on infeasible. Therefore we used the inductive approach and based our decision on the principles of object orientation, more specifically, the encapsulation principle. In the object oriented approach a program is organized as data (held in attributes) surrounded by methods that implement any manipulation on these data. Therefore, we expect that targeting the state-dependant fault type would allow us to indirectly locate other types of faults as well is it should be contained in the propagation chain if the encapsulation principle is probably adhered to. The experimental evaluation in chapter 5 supports this claim, as we were able to locate all the 50 faults in the experiment even though they were of different types.

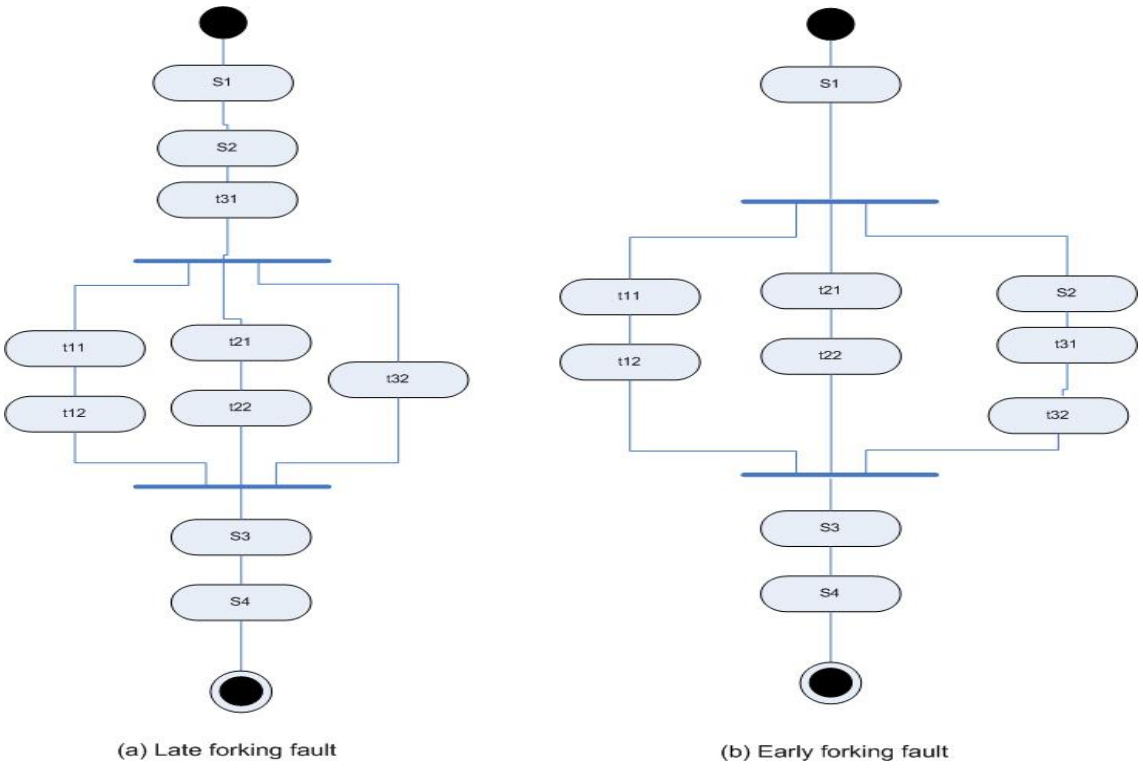

Figure 3.9: Forking faults examples

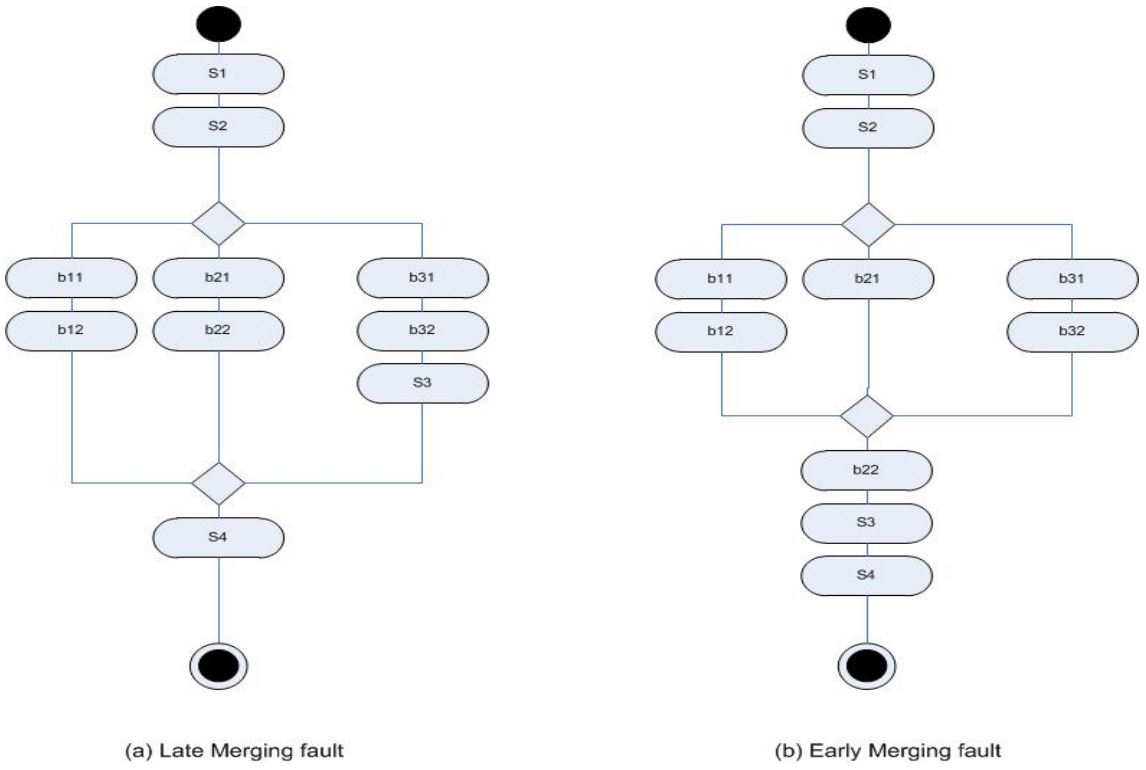

Figure 3.10: Merging faults examples

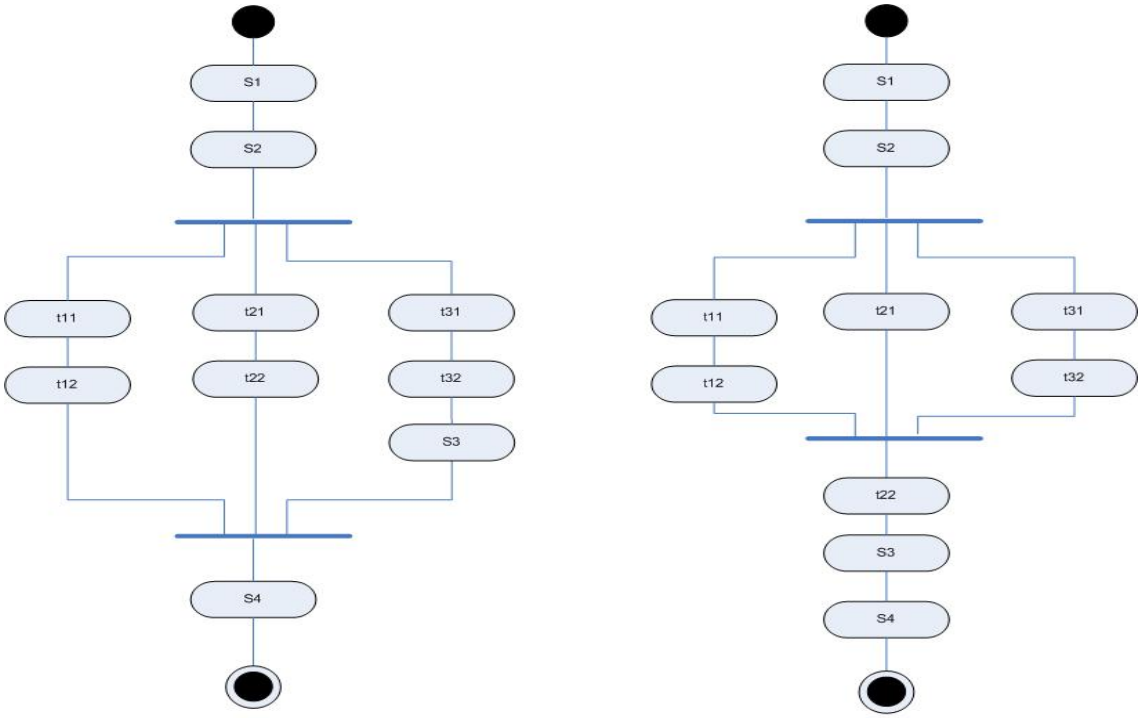

Figure 3.11: Joining faults examples

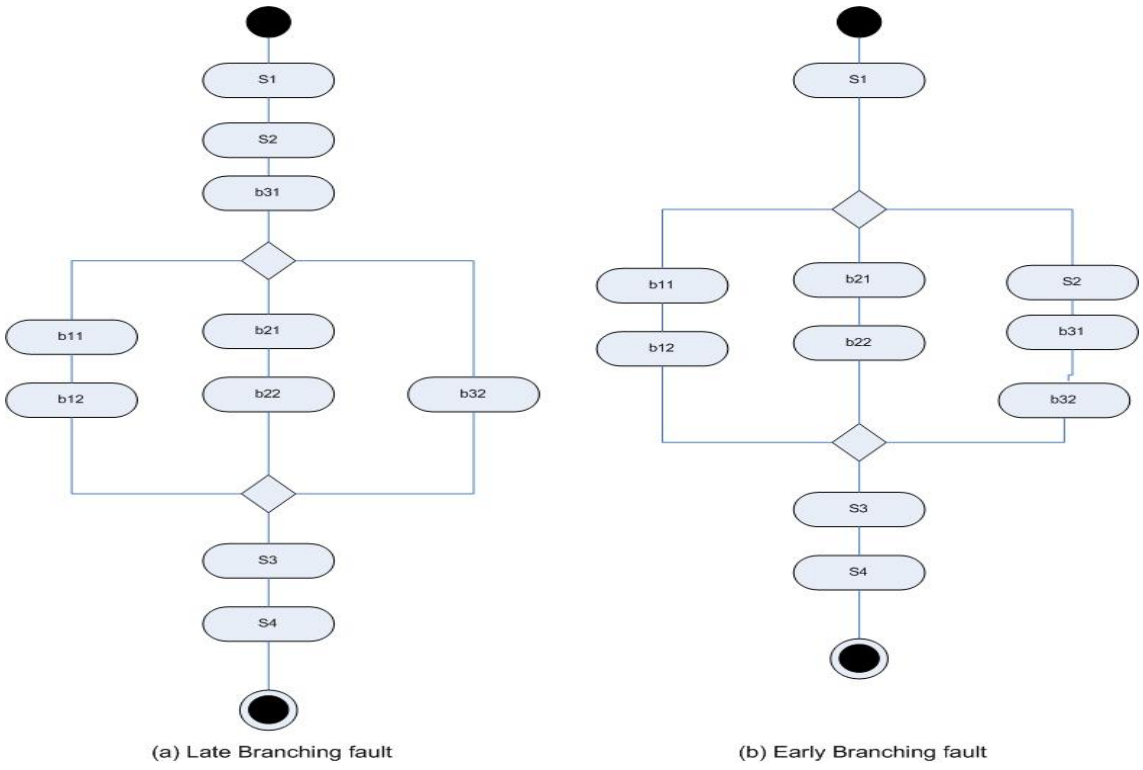

Figure 3.12: Branching faults examples

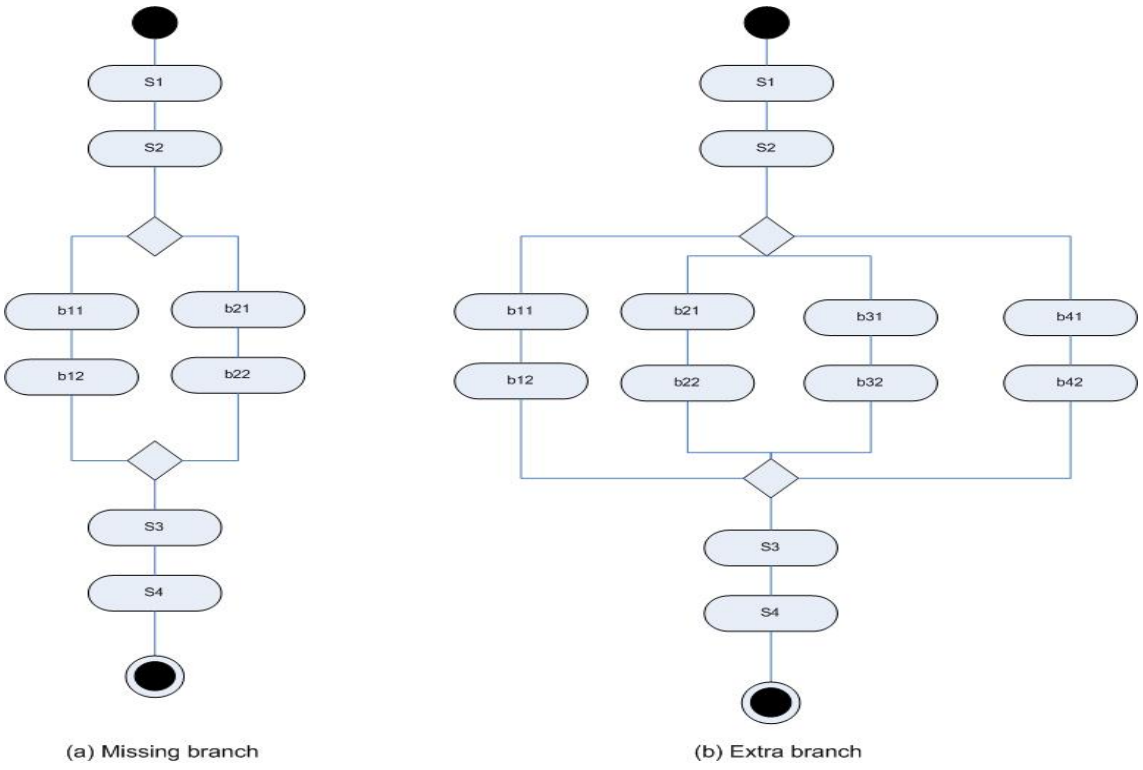

Figure 3.13: Branches faults examples

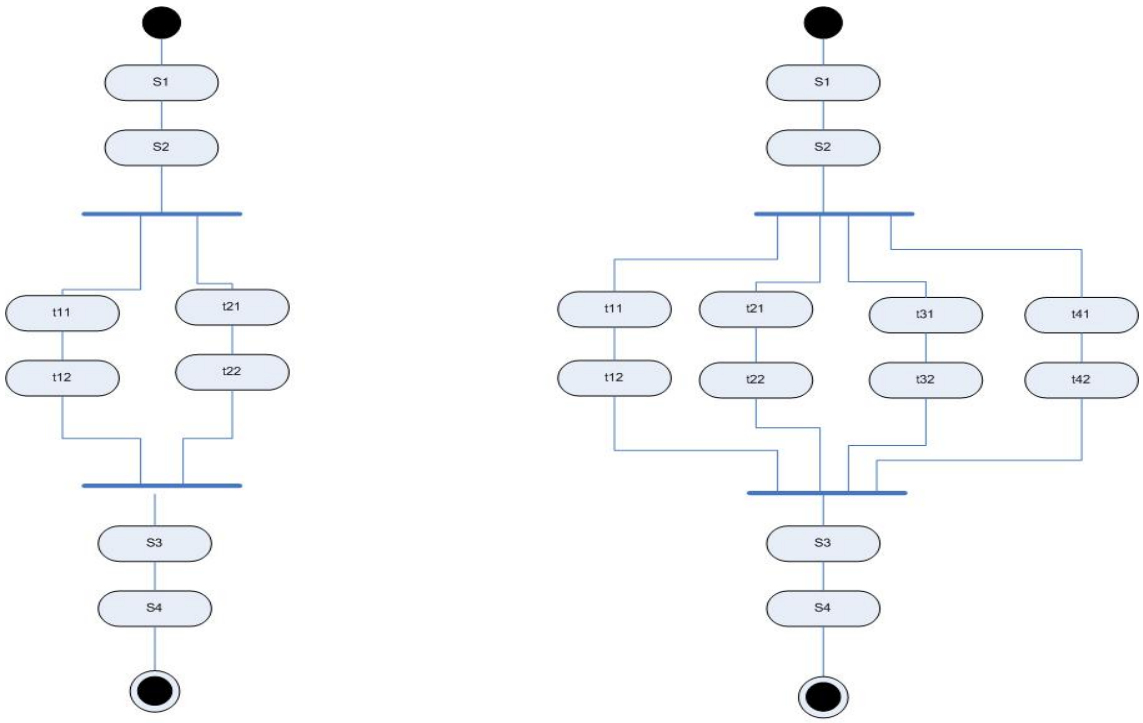

Figure 3.14: Tines faults examples

Chapter 4

Applying an Off-the-Shelf Tool

Trying to explore the possibility of using already existing data mining techniques to solve the software fault localisation problem, we considered the localization problem as a classification task and applied five of the classification algorithms implemented in Weka, an open source data mining tool. We published the results of this experiment applied on thirteen different faults of a C++ subject program named Concordance in the Automated Software Engineering (ASE) conference 2009 [4]. In this chapter, we explain the details of the published results as well as replicating it on several seeded faults in Java subject programs.

4.1 The Weka Tool

Weka [1] is an open source Java program provided by the machine learning group at the University of Waikato that implements several data mining algorithms for classification, clustering and association mining as well as for data pre-processing and visualization. It provides means to access those algorithms directly through a graphical user interface and a command line interface, in addition to the ability to call them from within Java code.

[1] http://www.cs.waikato.ac.nz/ml/weka/

4.2 Study Design

When we started on this experiment, we had only two variables in mind. One of them was the fault, as we had 13 different faults seeded in the concordance subject program. The other variable was the classifier, as we aimed at comparing the different classifiers implemented in Weka to see which of them can be most useful in the fault localization problem. A second stage of the experiment started after we observed in the first stage that the results suffered from the class-imbalance problem (see below). Therefore we repeated the experiments after wrapping the classifiers in a cost sensitive algorithm, which added a third variable to the experiment, that is, the cost-sensitivity of the classifier. In addition to that, we want to see whether adding black-box information to about the test cases would help in the classification, which is considered the fourth variable in the experiment. Finally, in a third stage of the experiment we wanted to see how the classifiers would perform in localizing faults seeded in Java subject programs that we used to evaluate the proposed "state-dependent" fault localization technique. The test suites of these programs consisted of unit test cases.

4.2.1 Subject Programs

Concordance

Concordance is a utility for making concordances (word indices) of documents. The original program was written in C++ by Ralph L. Meyer, and made available under the GNU General Public Licence on various open-source websites. The original program consists of 2354 lines of C++ code, spread over four .cc (primary source code) files and four .h (header) files.

To use the program, the user names an input file on the command line, and specifies whether they want an index by line, page or stanza. The program creates and writes to a file containing the index, and another file containing a table of letter frequency counts. The user can give the program various options, telling it what line, page or stanza to start counting on, the prefix of the output files, and whether to work in "quiet" mode.

Java programs

Information about these subject programs is available in chapter 5.

4.2.2 Faults

For concordance we had 13 different faults that occurred naturally in the subject program, and were discovered by students of a fourth year testing course at the computer science department at the University of Western Ontario. For the sake of being used in experiments, each of the discovered faults and its fix were allowed to be set on or off using C pre-processing variables; more information about the subject is available in Ali et al. [4]. For our experiment, we only allowed one fault to be "ON" at a time, which allowed us to have 13 different faulty versions of the subject program. Table 4.1 shows the types of each of these faults classified according to the taxonomy suggested in chapter 3, as well as the number of test cases that fail due to the activation of that fault out of the 372 available test cases. For the third stage of the experiment, we used the 50 faults used in evaluating the state-dependent fault localization process explained in chapter 5, where information about these faults and their subject programs are provided.

4.2.3 Data Collection

For each of the 13 faulty versions, we collected the data on coverage, success and failure for each test case into a file, one file per version. Each line of the file represented one test case. The first column was the test case number; the next 1490 columns represented the 1490 lines from the source file and the last column was either S, if the test case was successful, or F, if it was failing. We eliminated the code lines that had exact same behavior in all the test cases, as they would not affect the process of distinguishing between successful and failing behaviors. For each test case row and source line number, the value was the number of times the test case executed the line.

In data mining terminology, the version data file therefore represented 372 instances (one

Fault No.	# of Failing tests	Fault Type
1	24	2.2.3.1 Missing call fault
2	10	2.1.1.4.2. Extra branch
3	1	1.1.2.2.2. Multiplicity Below minimum fault
4	2	2.1.1.4.1. Missing branch (*2) [2]
5	34	2.1.1.3.2. Late merging fault
6	59	2.1.1.4.1. Missing branch (*n) [3]
7	57	2.1.1.1 Logical condition fault
8	16	2.1.1.4.2. Extra branch
9	17	2.1.1.1 Logical condition fault 2.2.1.1.2.2. Faulty argument value(s) (*2)
10	2	2.1.1.4.2. Extra branch (*n)
11	3	2.1.1.4.1. Missing branch
12	2	2.2.1.1.2.2. Faulty argument value(s) 2.1.1.1 Logical condition fault 2.1.1.4.1. Missing branch
13	3	2.1.1.4.2. Extra branch

Table 4.1: Faults seeded to Concordance

per test case) in two classes (success and failure). A commonly-used technique in data mining is classification: finding which features of instances predict which class the instance is in. A classifier algorithm takes instances and yields a method of predicting, from the instance data given, which class the instance is in.

For the Java subject programs, we utilized the behavior data that we collected for the state-dependent fault localization process that only recorded the first line of any executed conditional block. We think that this should be enough, as all the lines in a conditional block would have similar behavior to the first line in the block.

4.2.4 Application of Weka

Taking our problem as a classification problem, we applied Weka [15] to it. Weka implements classification algorithms of many different types; for example, the 13 "tree" classifiers output decision trees. We chose to focus on the 10 rule-based classifiers, since they output a model in the form of a rule (usually, a conjunction of first-order logic implications) that can be applied

directly to fault localization.

We could not use five of the rule-based classification algorithms for various reasons: for instance, the M5Rules classifier cannot handle binary classes (like our success/failure classes), and "Prism" cannot handle numeric attributes like our number of executions data. We were left with five classifiers: ConjunctiveRule, JRip, OneR, PART, and Ridor.

We applied each of the five classifiers to the data files of each of the 13 versions. For each classifier, we recorded the rule yielded by the classifier, and the number of successful and failing test cases that were correctly and incorrectly classified by the rule.

Data mining classifiers are generally stochastic and do not necessarily attempt to classify with complete accuracy. We measured the performance of the classifiers using three standard metrics: precision, recall and accuracy. For FL, we are interested in giving the programmer information characterizing failing test cases, so we define precision as the percentage of instances that the rule classifies as failing that are actually failing; recall as the percentage of failing instances that the rule classifies as failing; and accuracy as the harmonic mean of precision and recall, which is the standard summary measure combining precision and recall.

Our initial findings did not look encouraging (the quantitative results are discussed below). Precision was sometimes good, but recall was usually poor. Many of the classifiers yielded a rule that simply classified all instances in the "success" class; this apparently seemed reasonable to the classifiers because most of the instances were successes and therefore the total percentage of correctly-classified instances was high.

We came up with two working hypotheses as to why the Weka classifiers performed so poorly. The first was that the data set suffered from the class imbalance problem [22]. This is the characteristic that a data set has if one class is much larger than the others. It becomes a "problem" for data mining because the usual assumptions of data mining do not apply well in such cases. Our data, like most test case success and failure data, contained one large class (the successful test cases) and one class (the failing test cases) that was much smaller relative to the other class.

The other working hypothesis was the absence of blackbox information about the test cases. For many of the faulty versions, some of the failure instances were very close to, or indistinguishable from, success instances. As an artificial example of how this can occur, consider a program in which the faulty passage of code is *y=10;*, whereas the correct passage of code is *if (x!=10) y=10;*. The faulty program manifests a failure only if *x==10*, but in this case it may execute exactly the same code as every successful test case. Our hypothesis was that adding information about the black-box (requirements-based) intent of the test cases might serve to differentiate between successful and failing test cases that executed the same code, and thus help the classifiers find a more accurate classification rule.

Because of our initial findings and our working hypotheses, we performed an experiment to find whether compensating for the class imbalance problem, or adding black-box information, or both, helped the classification.

There have been various approaches proposed for compensating for the class imbalance problem [22], but only two are implemented in Weka. One, MetaCost, chooses to predict the class with the minimum expected misclassification cost, whereas we always want to predict the failure set. We therefore used the other Weka algorithm, "CostSensitiveClassification", which allowed us to assign a cost to misclassification of instances. We assigned each successful instance a cost of 1, and each failing instance a cost of (number of successful instances) / (number of failing instances). This guaranteed that the total cost of all successful instances was equal to the total cost of all failing instances.

To add black-box information, we classified each test case for concordance according to 10 black-box testing characteristics, each of which could have 3 to 13 distinct numeric values. For instance, characteristic 7 was contents of input file; the value 1 for this characteristic indicated an empty input file, 4 indicated that a very long word occurred in the input file, and 11 indicated that no input file name was given or the file was nonexistent. After assigning black-box characteristics, we added 10 columns to the data files, containing the numeric values of each of the characteristics for each of the test cases. This black-box information thus became an extra

set of features that the classifiers could use in classification.

We then repeated our classification study. For each of the four combinations of (black box information, no black box information) and (imbalanced cost, balanced cost), we applied each of the five Weka classifiers to each of the 13 data sets, and recorded the resulting precision, recall and accuracy. The total number of observations was therefore 2 x 2 x 5 x 13 = 260 for each of precision, recall and accuracy. We identified 10 different groups of values for the black-box information, based on the syntax of the command line usability of concordance. Each test case is assigned exactly one value for each of the groups taken out of its possible values. Table B.1 shows the descriptions of the 10 groups as well as their possible values.

For the third stage of the experiment the black-box information was not relevant since we only had white-box test suites. Hence we had additional 2 x 5 x 50 = 500 observations. That is, for each of the 50 faults we applied the 5 classification algorithms once by itself and another time after wrapping it in the cost-sensitive algorithm.

4.3 Study Results

Figures 4.1 through 4.7 show box-plots for the accuracy, precision and recall of all the observations on all the subject programs, as well as for some of the individual subject programs. The cost sensitivity factor seemed to be significant in improving the values of the measures as shown in Figure 4.2.

In general, all classifiers could be considered poor when used as fault localizers, even though the cost-sensitivity did provide some improvement on the recall and the accuracy. Among the five classifiers, ConjunctiveRule seemed to be the worst, while PART was the best in recall, and OneR tied with PART in accuracy.

Figure 4.3 provides an example that shows the variation in the accuracy of the different classifiers, with PART being the best among them. Figure 4.7 provides an example of the unpredictability of the classifiers, as it goes against the general observation, since Conjunc-

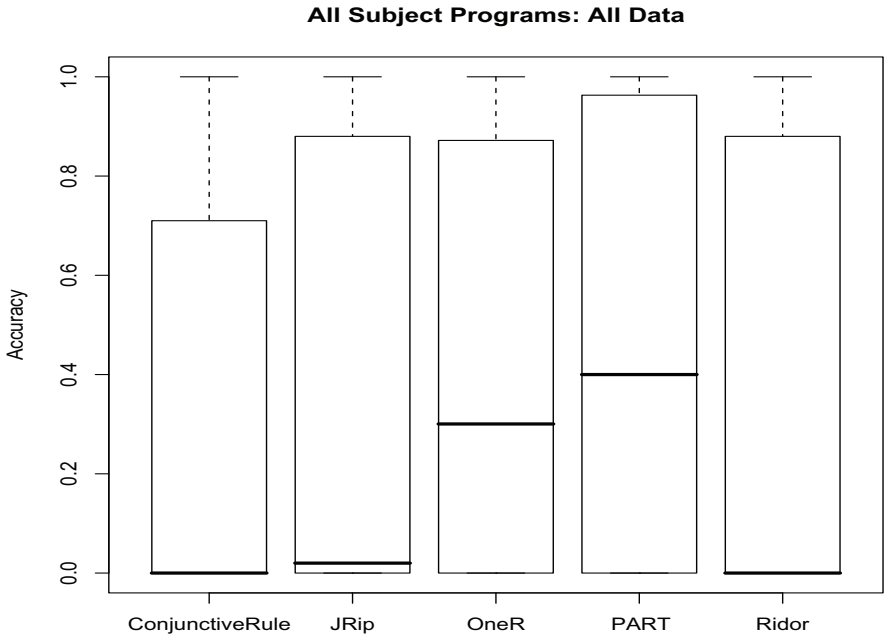

Figure 4.1: Accuracy of classifiers, all subject programs.

tiveRule has the best value and OneR is actually better than PART.

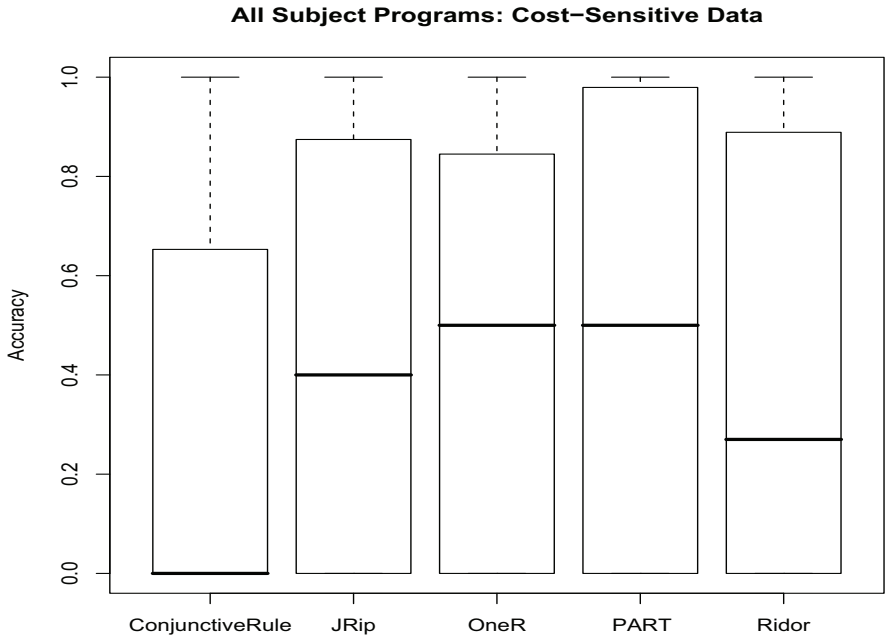

Figure 4.2: Accuracy of classifiers, all subject programs, cost sensitive data only.

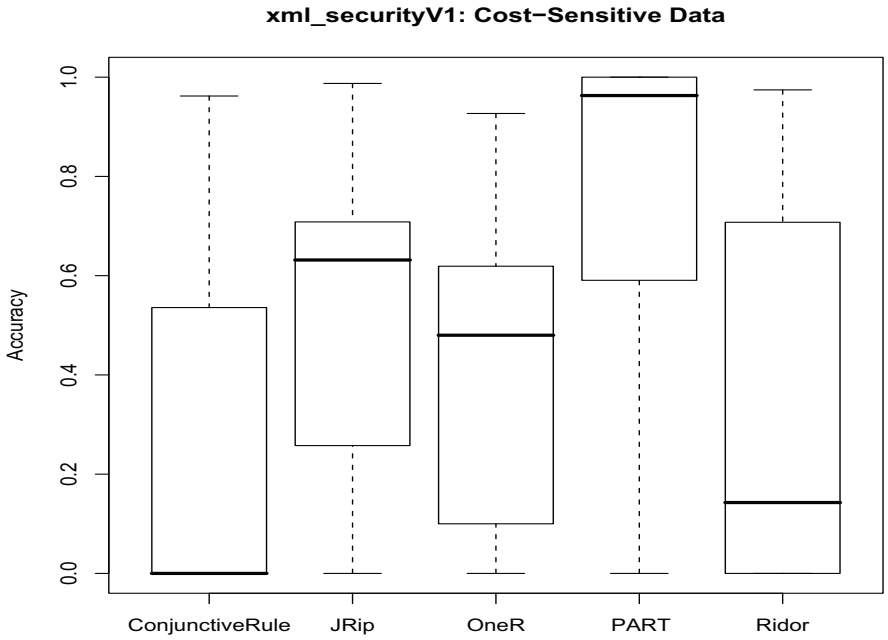

Figure 4.3: Accuracy of classifiers, XML_SecurityV1, cost-sensitive data only.

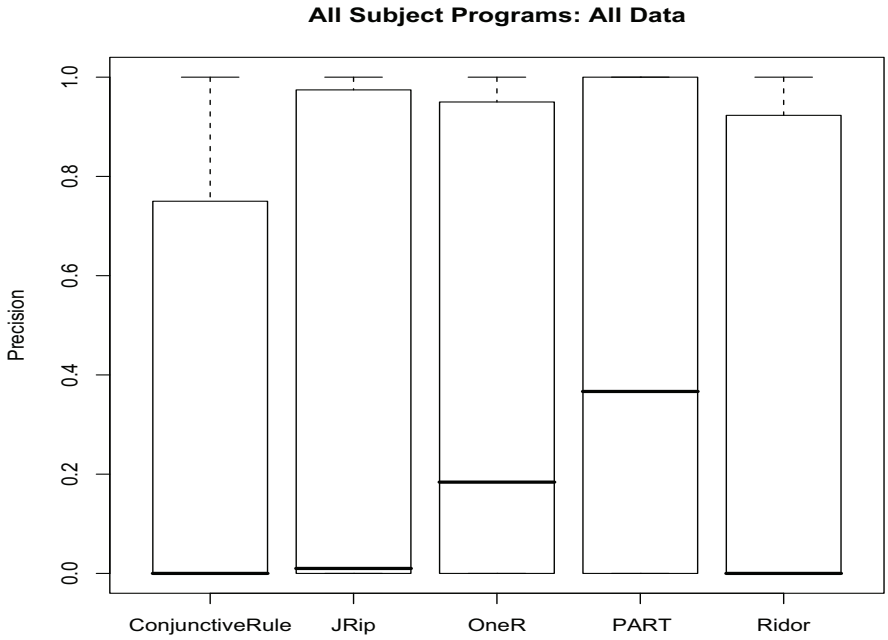

Figure 4.4: Precision of classiferies, all subject programs.

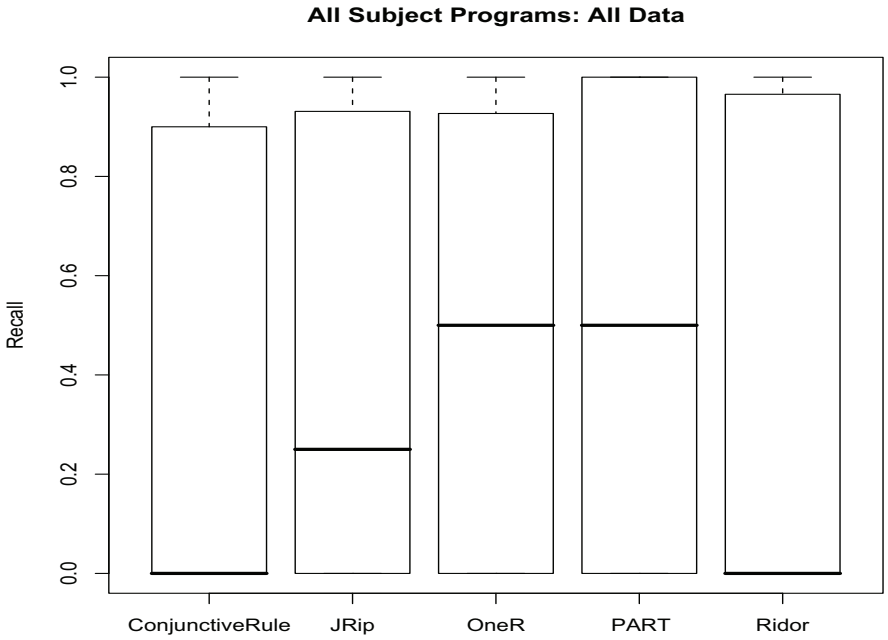

Figure 4.5: Recall of classifiers, all subject programs.

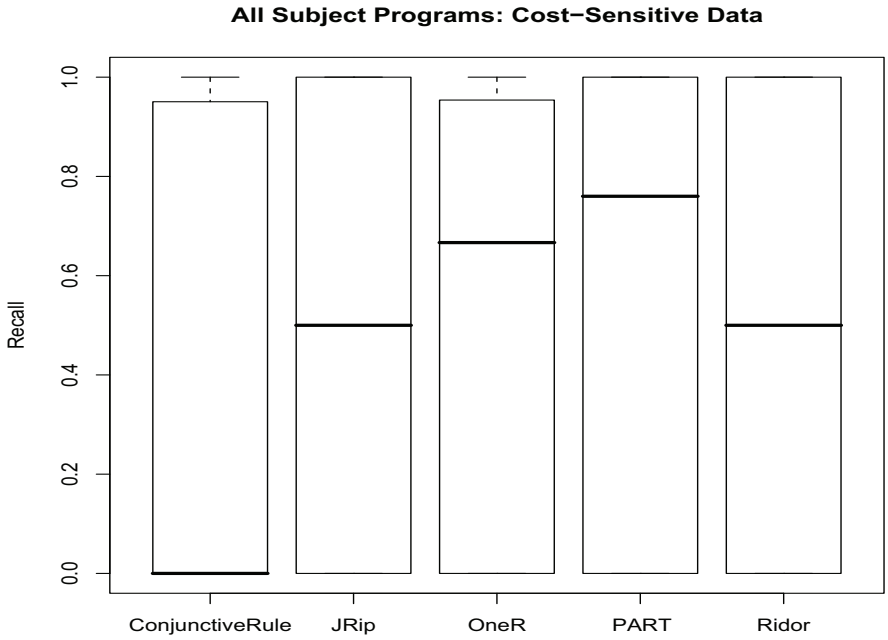

Figure 4.6: Recall of classifiers, all subject programs, cost-sensitive data only.

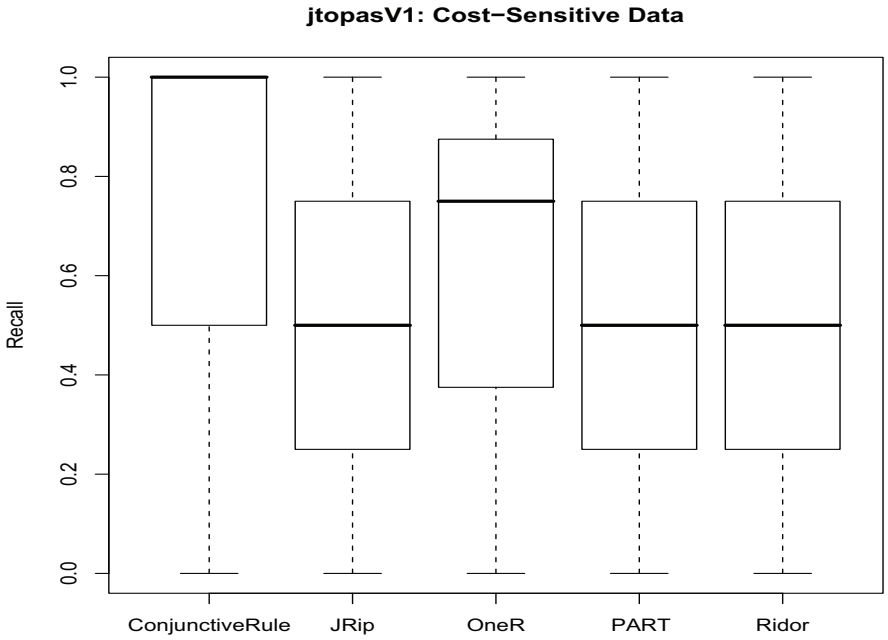

Figure 4.7: Recall of classifiers, subject program JTopas version 1, cost-sensitive data only.

Subject Program	Recall	Precision	Accuracy
concordance	0.6994	0.4597	0.5086
jmeterV4	0.07556	0.0925	0.08291
jmeterV5	0.4552	0.3697	0.3998
jtopasV1	0.55	0.3931	0.4186
jtopasV2	0.5	0.47305	0.4559
jtopasV3	0.168	0.1330	0.1423
xml_securityV1	0.5204	0.4294	0.4602
xml_securityV2	0.9503	0.9480	0.9490
xml_securityV3	0.7648	0.7464	0.7513
All Subjects	0.4876	0.3978	0.4177

Table 4.2: Average recall, precision and accuracy of the PART classifier, across all subject programs, under the cost-sensitive treatment.

Subject Program	Recall	Precision	Accuracy
concordance	0.7954	0.5677	0.5962
jmeterV4	0.08889	0.1	0.09412
jmeterV5	0.4887	0.3864	0.4266
jtopasV1	0.50	0.4444	0.4667
jtopasV2	0.6667	0.6667	0.6667
jtopasV3	0.16	0.16	0.16
xml_securityV1	0.7917	0.6998	0.7349
xml_securityV2	0.9935	0.9907	0.9920
xml_securityV3	0.7605	0.8125	0.7853
All subjects	0.5582	0.4786	0.4959

Table 4.3: Average recall, precision and accuracy of Weka classifiers, across all subject programs, under the cost-sensitive treatment.

When the class imbalance problem was not taken into account, the average recall over all subject programs and classifiers was 0.3392, the average precision was 0.3613, and the average accuracy was 0.346. However, when the class imbalance problem was taken into account by assigning the higher cost to misclassifying failures, the average recall rose to 0.4876, precision to 0.3978, and accuracy to 0.4177. For each of recall, precision and accuracy, we performed a Wilcoxon signed rank test between the cost-insensitive and cost-sensitive treatments, finding that they were statistically significant for all three measures ($p < 0.001$, $p = 0.01703$, and $p < 0.001$ respectively).

Table 4.2 shows the mean values for recall, precision and accuracy across all classifiers for the various subject programs, under the cost-sensitive treatment. The performance of the Weka classifiers varied widely depending on the subject program. For instance, they performed very well on `xml-security` version 2, with almost all data points at 1 for recall, precision and accuracy; however, they performed very poorly on `jmeter` version 4 and `jtopas` version 3, with 75% of the data points at 0 for all three measures.

PART had the best average recall, precision and accuracy out of all the classifiers. Table 4.3 shows the values of recall, precision and accuracy for just PART, across all subject programs. However, usually the differences between classifiers were not statistically significant. We performed a Tukey Honest Significant Differences (HSD) test on all the pairs of the five classifiers, for recall, precision, and accuracy, and for all the data together and for each subject program individually. As shown in tables 4.4 and 4.5 we found that usually, no pair of classifiers showed a statistically significant difference overall, but that sometimes PART was significantly better than ConjunctiveRule in precision and accuracy.

4.4 Summary

In summary, the observations that we found in the first two stages of the experiment on the concordance subject program, that the mechanism implemented on the classifier as well as the nature of the fault to be localized were significant factors in the usability of classification algorithms in fault localization, were re-enforced by the results of applying the same experiment on Java subject programs in the third stage of the experiment. In general, some of the classifiers did well on some of the faults; however, in practice we cannot recommend any of the classifiers to be used by programmers as it depends on the fault to be found, which we cannot determine beforehand.

Pair	Concordance	Jtopas V1	Jtopas V2	Jtopas V3	XML Security V1	XML Security V2	XML Security V3	JMeter V4	JMeter V5
JRip-ConjunctiveRule	N	N	N	N	N	N	N	N	N
OneR-ConjunctiveRule	N	N	N	N	N	N	N	N	N
PART-ConjunctiveRule	Y	N	N	N	N	N	N	N	N
Ridor-ConjunctiveRule	N	N	N	N	N	N	N	N	N
OneR-JRip	N	N	N	N	N	N	N	N	N
PART-JRip	N	N	N	N	N	N	N	N	N
Ridor-JRip	N	N	N	N	N	N	N	N	N
PART-OneR	N	N	N	N	N	N	N	N	N
Ridor-OneR	N	N	N	N	N	N	N	N	N
Ridor-PART	N	N	N	N	N	N	N	N	N

Table 4.4: Pair-wise significance of the precision

Pair	Concordance	Jtopas V1	Jtopas V2	Jtopas V3	XML Security V1	XML Security V2	XML Security V3	JMeter V4	JMeter V5
JRip-ConjunctiveRule	N	N	N	N	N	N	N	N	N
OneR-ConjunctiveRule	N	N	N	N	N	N	N	N	N
PART-ConjunctiveRule	Y	N	N	N	N	N	N	N	N
Ridor-ConjunctiveRule	N	N	N	N	N	N	N	N	N
OneR-JRip	N	N	N	N	N	N	N	N	N
PART-JRip	N	N	N	N	N	N	N	N	N
Ridor-JRip	N	N	N	N	N	N	N	N	N
PART-OneR	N	N	N	N	N	N	N	N	N
Ridor-OneR	N	N	N	N	N	N	N	N	N
Ridor-PART	N	N	N	N	N	N	N	N	N

Table 4.5: Pair-wise significance of the accuracy

Chapter 5

State-Dependent Fault Localization Process and a Customized Algorithm

In this chapter we present the process of localizing state-dependent faults, as well as a customized mining technique to be used to find pure failing behaviors. Both the localization process and the mining technique are experimented on and evaluated; the experimental design and results are explained here as well.

Section 5.1 gives an overview of our proposed fault localization process. Section 5.2 summarizes the goals of the key algorithm used in the fault localization process, namely the associated sequence mining algorithm. This algorithm is based on the earlier FP-Growth algorithm, which is summarized in section 5.3. Section 5.4 gives details of the proposed new algorithm. Section 5.5 describes an experiment for evaluating the runtime performance of the new algorithm. Section 5.6 describes an experiment for evaluating how useful the new fault localization process is. Finally, section 5.7 summarizes the chapter.

5.1 State-Dependent fault localization process

Figure 5.1 illustrates the proposed process of localizing state dependent faults. The process consists of six main steps, namely, pre-instrumentation, instrumentation, running test suites,

splitting (which consists of three sub-steps as illustrated in the figure), finding pure fails, and finally hierarchical ranking. Each of these steps and the intermediate artifacts are explained below.

Step 1: Pre-Instrumentation

In order to perform state-dependent fault localization, we need to collect the state-dependent behavior of all test cases. This includes not only information about what lines of code were run, but also the values of variables (Java fields) in objects at the time that object methods were run.

Instrumenting the source code to collect the state-dependent behavior would involve adding code lines to log information (as illustrated in step 2) in order to be analyzed. In order to make sure that the added lines will not change the behavior of the program or cause compilation errors, both the source code files and test cases are pre-instrumented by removing all comments and then automatically formatting it to insure consistency of its organization. To remove the comments, an already existing Java class was downloaded from an online forum [1] and used after fixing some bugs related to dealing with nested comments. The auto formatting tool provided by Eclipse was used to insure consistent format of the code.

Step 2: Instrumentation

In order to collect the state-dependent behavior, a header is added to the top of each method to print the name of the method and the names and values of all the primitive member variables of that class. Non-primitive member variables are not considered because they conceptually are implementations of associations, which are part of the system state, not the object/class state that we are focusing on. Since static methods cannot access the member variables, only static members' names and values are printed in the header of static methods. However, the opposite

[1] http://onlinefundb.com/forum/index.php?topic=2531.0

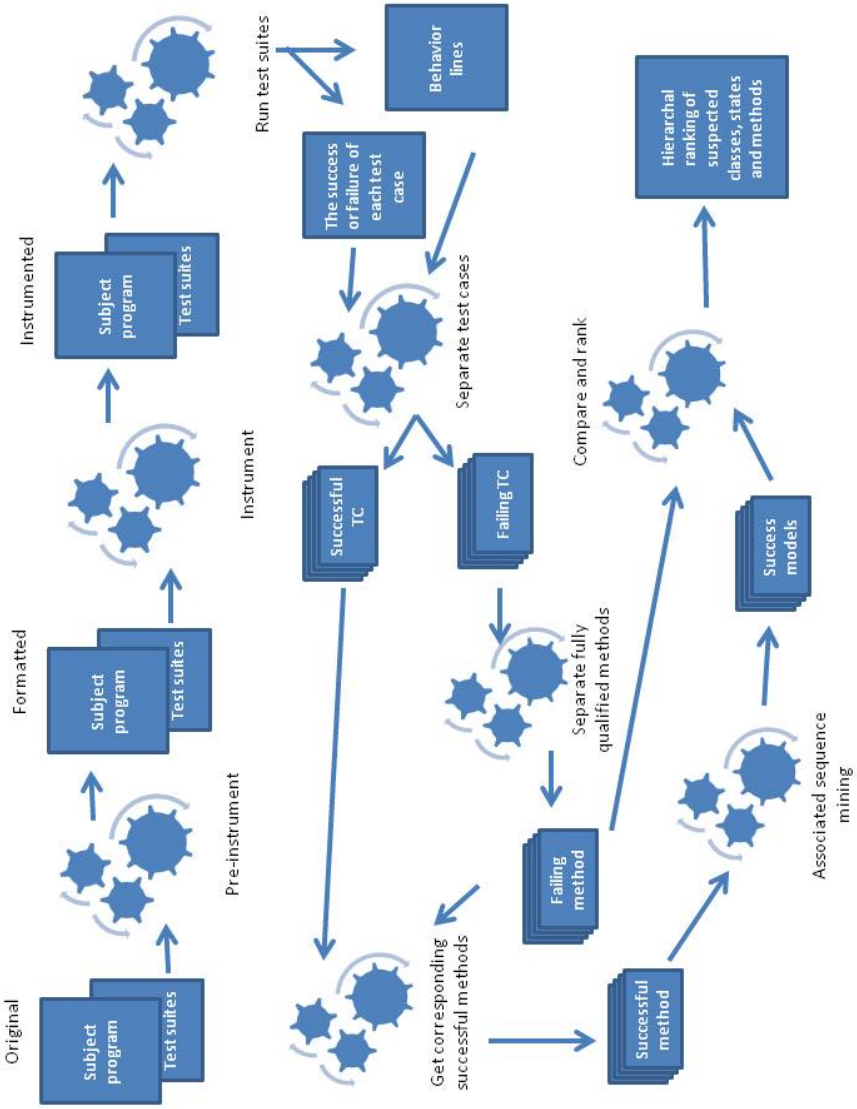

Figure 5.1: State-dependent faults localization process

is not true because non-static methods can access static member variables, since they contain one value per class that is shared across all objects.

With the header of the method, the number of the last line in the header is printed to represent the execution of that method at a specific state. The behavior of the method is represented by printing the line number at the beginning of each conditional block. Note that the line numbers are not important for themselves, since we have amended the organization of the source code in the pre-instrumentation step. However, they can only be used in comparison to other behavior lines to highlight the suspicious behavior. For the sake of proper organization, the name of the method preceded by the fully qualified name of the class is printed in the beginning of the behavior line.

Here we had to make a decision for inherited methods, whether to print the name of the subclass through which it was called or the superclass where it was defined. Even though printing the name of the superclass would be more accurate for localization, it would lead to loss of information, as knowing the context in which a method was executed would be important to determine why it failed and how to fix it. Therefore, we decided to dynamically print the class according to the object type the method is called through during run-time. This is not the same for static methods; since they are not dependent on the instances, the name of the class where they are defined is printed with them.

We also instrumented the test suites. However, the instrumentation of the test suite is just by printing the name of the test method in order to be able to isolate the behavior related to it in further steps.

Step 3: Running Test suites

In this step, all instrumented test suites are run on the instrumented subject program, and the behavior information is collected in one behavior log that has the names of test cases as headers followed by the behavior lines of methods executed under each test case. In addition to that, a report indicating which of the test cases were failures is also maintained, in order to be used to

separate failing behavior from successful behavior. Table 5.1 shows an excerpt of the behavior log collected while running the test cases on XML-security version 1 after being seeded by one of the studied faults. The line surrounded by %s is the line printed by the instrumented test method and highlights the start of the execution of code related to that test case. The following lines are lines printed by the instrumented source code methods.

Step 4: Splitting

The first type of split we need to do is to split the behavior according to whether the test case that caused that behavior failed. The rest of the behavior logs are considered successful, except for those lines executed during the set up or teardown of each test suite. We cannot associate these lines to a specific test case; therefore they are isolated as neutral and will not be considered in the comparison to avoid confusion. Since a test case may have caused methods of different objects to be called, we need to perform another split, to collect behavior of each method in a separate file organized in a directory for each class.

Step 5: Finding pure fails

To identify the pure failing behavior, we compare each method's failing behavior to the corresponding successful behavior. However, comparing the failing behavior to the raw successful behavior might lead to confusion, since several failing behaviors might be exhibited, less frequently, in the successful log. Therefore, a behavior will not be considered successful unless there is enough evidence that it should be identified as successful. That evidence is provided by the success models generated by applying the associated sequence mining technique. A behavior line is removed from the pure failing behavior only if its state was identified in the successful rules in association sequence(s) of statements similar to the sequence(s) logged in the failing behavior. The success rules are calculated using the customized mining algorithm illustrated in section 5.2.

%%%%% org.apache.xml.security.test.c14n.implementations.Canonicalizer20010315Test-test31withComments() %%%%%
org.apache.xml.security.utils.IgnoreAllErrorHandler-error(SAXParseException ex), warnOnExceptions=true,throwExceptions=false# L74,L79,
org.apache.xml.security.utils.XMLUtils-circumventBug2650(Document doc)#L1493,
org.apache.xml.security.utils.XMLUtils-circumventBug2650recurse(Node node)#L1513,L1555,L1555,L1555,
org.apache.xml.security.utils.XMLUtils-circumventBug2650recurse(Node node)#L1513,L1519,L1555,L1555,L1555,L1555,L1555,
org.apache.xml.security.c14n.Canonicalizer-getInstance(String algorithmURI), _alreadyInitialized=true#L96,
org.apache.xml.security.c14n.Canonicalizer-getImplementingClass(String URI), _alreadyInitialized=true#L447,L455,L455,L455,L455,
org.apache.xml.security.c14n.Canonicalizer-canonicalizeSubtree(Node node), _alreadyInitialized=true#L258,

Table 5.1: An excerpt from the behavior log of one of the seeded faults of XML-Security V1

Step 6: Hierarchical ranking

Finally, hierarchical ranking of the failing behavior is calculated based on the number of times a certain behavior line (see Table 5.1) appeared in the failing behavior log. Then the rank of the containing method is calculated as the total number of times the behavior lines belonging to that method appeared in the failing behavior log. Then the class rank is calculated as the summation of the ranks of all its methods.

5.2 The customized algorithm : Associated sequence mining

Let $I = \{i_1, i_2, ..., i_k\}$ and $S = \{s_1, s_2, ..., s_l\}$ be two sets of items. Let $DB =< T_1, T_2, ..., T_m >$ be a dataset of transactions, where each transaction T_i contains a set of items drawn from I and a set of ordered items drawn from S separated by a "#". A pattern P is a set of items that may occur in DB. If P is drawn from I then it is called an *item-set* pattern, and if it is a sequence drawn from S then it is called a *sequential-set* pattern. The *support* of P is the number of transactions in DB that contain P. P is called a *frequent pattern* if its support is no less than a minimum support threshold ξ . The term *frequency* is often used interchangeably with the term *support*.

The aim of associated sequence mining technique is to find rules of the form

$$\text{Itemset pattern} \rightarrow \text{sequential-set pattern}$$

As in association rules mining, in order for a rule R to be included in the result it must have a confidence not less than a minimum confidence threshold β.

If R is of the form $IF\ P_1\ THEN\ P_2$ ($P_1 \rightarrow P_2$), the *confidence* of R is defined as:

$$confidence(R) = \frac{support(P_1 \bigcup P_2)}{support(P_1)} \tag{5.1}$$

The problem of the associated sequence mining was developed with a certain application in mind. That is, for object oriented software fault localization, the aim was to produce rules that associate the state of the objects (as the itemset) that imply the sequence of statements executed

in a certain method. However, the technique is applicable on a wide range of problems; for example, to find rules that associate the properties of a website's user and the sequence of pages that he or she navigates through.

5.3 FP-Growth mechanism

The challenging step of finding association rules is to find the frequent patterns, that is, patterns that satisfy the minimum support threshold. Then, calculating the confidence of different rules that can be generated from each frequent pattern should be straightforward. Hence, the main focus in the literature related to association rules mining is on that step.

The trivial approach is to calculate the support for all possible patterns, that is, scan the entire dataset calculating the support of each element in the power set of items. If n is the size of the dataset and m is the number of items that can occur in the dataset, the time cost of this approach will be $O(n * 2^m)$. Given that n should be a very large number, as the definition for data mining suggests, and that it is exponential in the number of items, this is obviously infeasible.

Different mechanisms proposed in the literature to replace this trivial approach by more feasible ones work in two directions, *structuring* and *pruning*. In *structuring* the dataset is re-organized in a new data structure allowing faster calculations; for example Gardarin et al. [23] suggests putting the dataset in a bitmap data structure. In *pruning*, unnecessary calculations are eliminated based on some logical principle. For example, the Apriori mechanism prunes based on the principle "An infrequent pattern cannot be included in another frequent pattern" [3].

FP-Growth (Frequent Pattern - Growth) is one of the fastest mechanisms existing in the literature [26]. In fact, it utilizes both structuring and pruning. For structuring, an extended prefix-tree structure called the FP-tree is used to store the patterns as well as crucial quantitative information about their frequencies. For pruning, first the apriori principle is used to eliminate

all infrequent single items from the dataset and build the FP-tree only by frequent single-item patterns. Then the tree is mined recursively while pruning based on the principle: the frequency of a pattern in the entire dataset is equal to the frequency of part of that pattern in the subset of transactions containing the other part. That is, the frequency of $(P_1 \bigcup P_2)$ in the entire dataset equals the count of P_1 in the subset of transactions containing P_2.

The construction and the mining of the FP-tree are illustrated in Han et al. [26]. Below, we show how the original FP-Growth algorithm works, and discuss its applicability to sequences as well as sets. We then compare the two methods.

5.3.1 The construction of the FP-Tree

The first step for mining using the FP-Growth algorithm is to build the FP-Tree, i.e. to re-organize the dataset in a new structure that is both condensed and contains item frequency information that will reduce the calculations needed for further steps. Before building the FP-Tree, a first scan for the entire dataset is performed in order to count the frequency/support of each single item. Items that have frequency less than the minimum support are eliminated from any further processing. The remaining items are then ordered according to their frequency as a preparation for building the FP-Tree. A second scan of the dataset is then performed in order to put the transactions in the tree. To do so, each transaction is processed to eliminate all infrequent items and order the remaining items according to the frequency order, that is, the more frequent items are considered as a prefix for the less frequent ones. The transactions are then set in the tree so that they share common prefixes.

To illustrate the steps of building the tree, let us take Table 5.2 as an example dataset that we are interested in mining with minimum support equal to 11.

For the first scan:

1. The frequency of each item is counted as follows: $\{j = 10, i = 11, h = 12, g = 10, f = 16, e = 10, d = 11, c = 10, b = 11, a = 9\}$

Figure 5.2: The FP tree after inserting T1

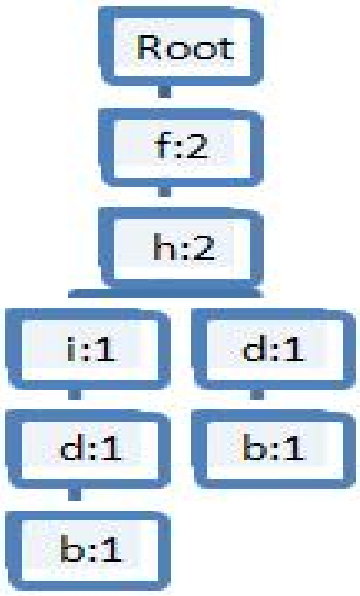

Figure 5.3: The FP tree after inserting T1 and T2

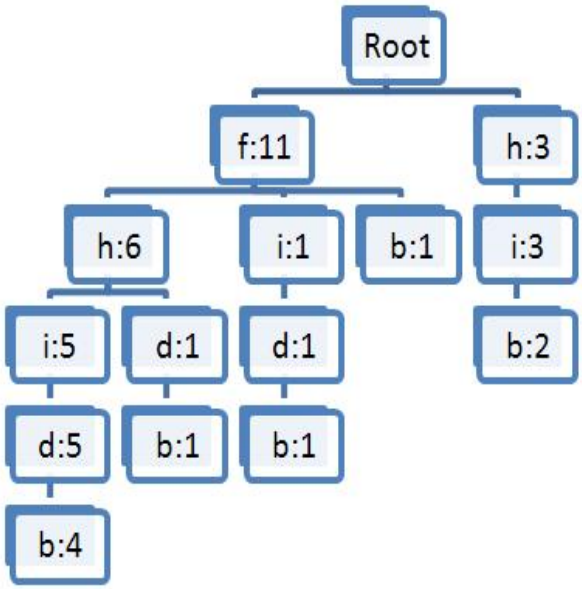

Figure 5.4: The FP tree after inserting transactions up to T14

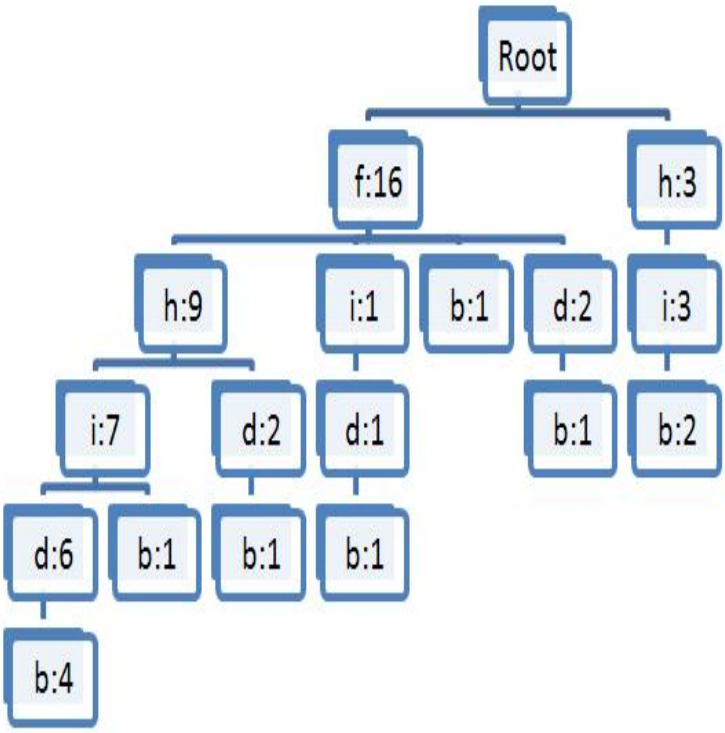

Figure 5.5: The FP tree after inserting transactions up to T19

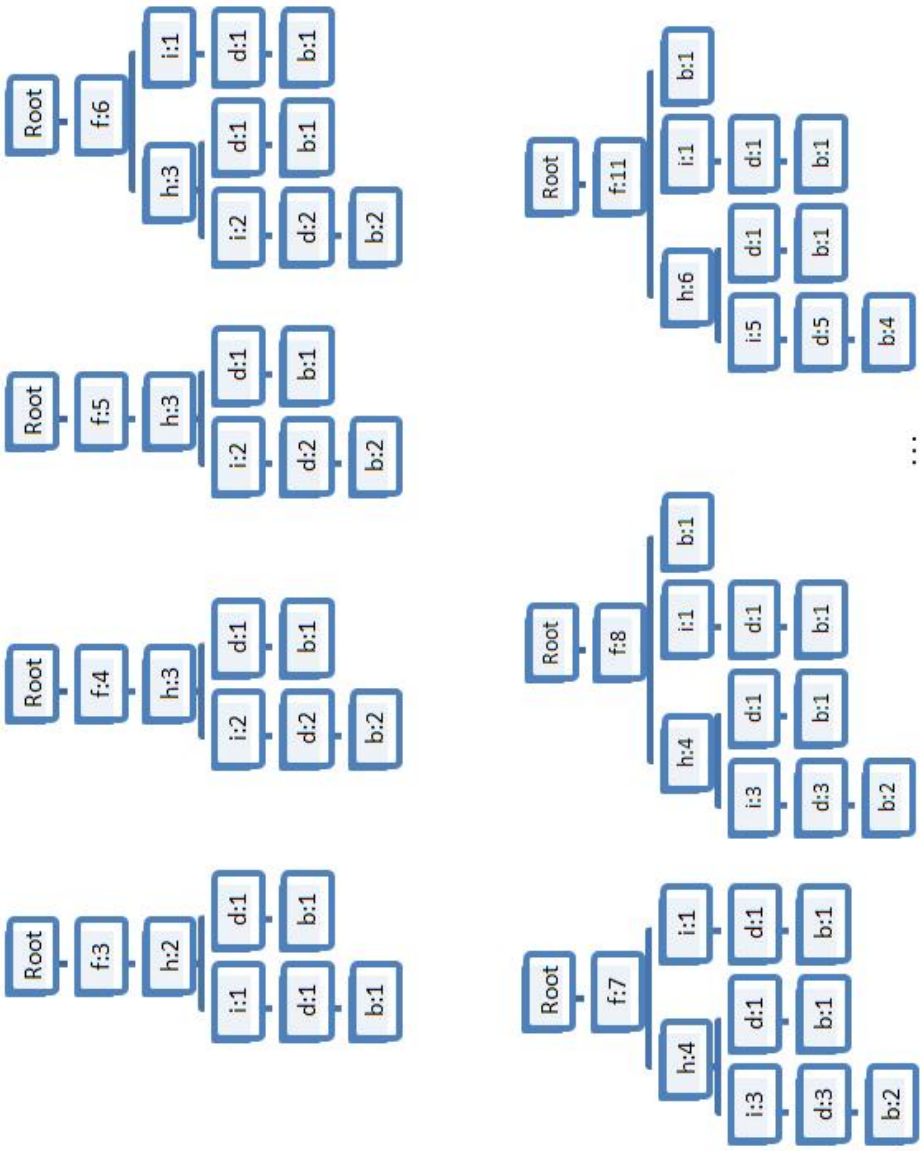

Figure 5.6: The FP tree after inserting transactions up to T11

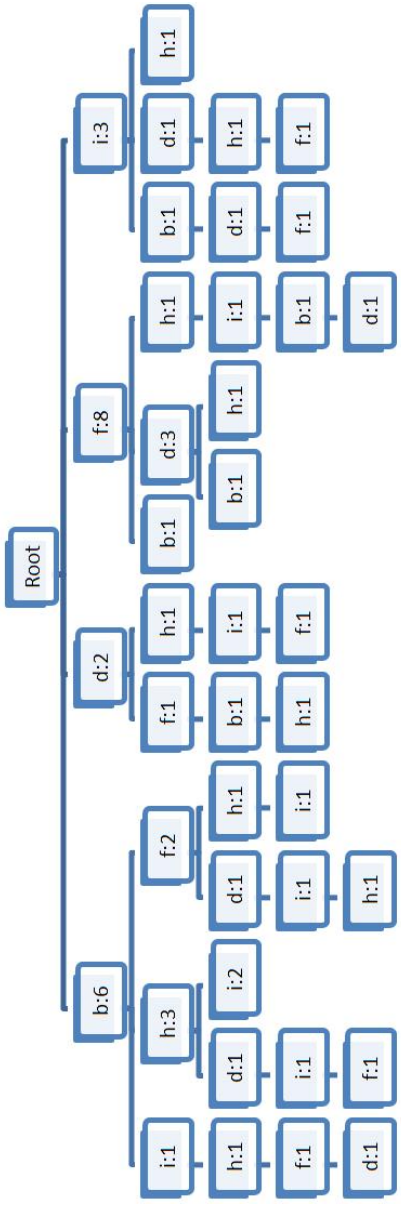

Figure 5.7: The FP tree with the order relationships preserved

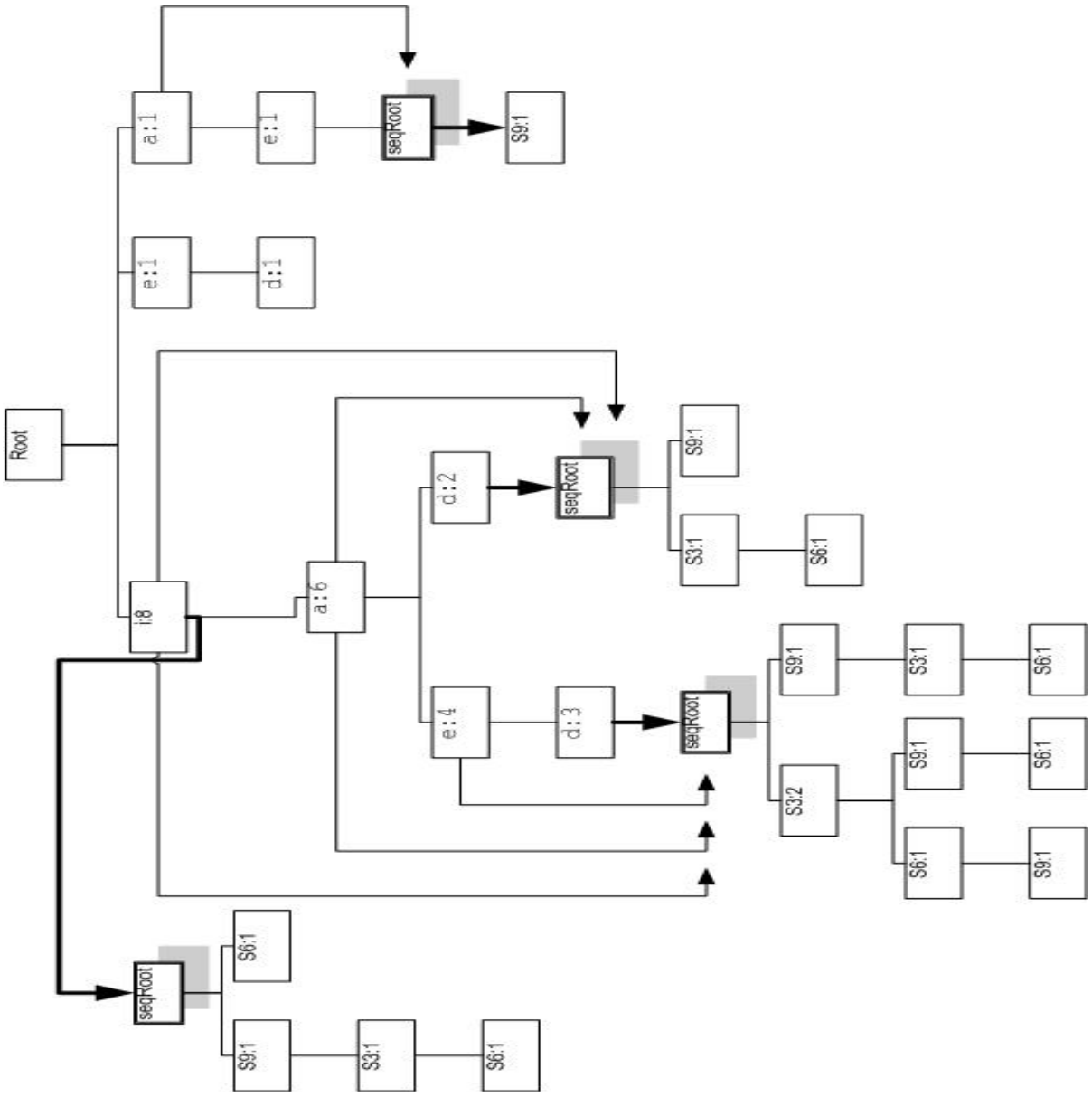

Figure 5.8: Mixed FP tree

2. Items having support less than 11 are removed and the items are ordered according to their support: $\{f = 16, h = 12, i = 11, d = 11, b = 11\}$

For the second scan:

1. Reading one transaction at a time, remove infrequent items and sort the remaining items according to the frequency calculated in the first scan.

2. The transactions after sorting should be as illustrated in the Table 5.3.

3. Then insert each transaction in the tree so that it shares a prefix with previous transactions.

   (a) Each node in the tree should contain the item that it represents and a number representing the number of transactions that contain the set of items that share the prefix represented by the path of the tree reaching that node.

   (b) Since there are no paths to share, the first transaction will start a new branch in the tree as shown in Figure 5.2.

   (c) The second transaction will share the prefix f,h with the first transaction, hence will increase the count of each of these nodes then start a new branch after h to add the d,b as shown in Figure 5.3.

   (d) Transactions up to T11 share at least f as a prefix, as they should be added to the tree as shown in Figure 5.6.

   (e) Transaction T12 does not share any prefix in the existing tree, hence a new branch will be started at the root for the item h, and transactions T13 and T14 will also be added to that branch as shown in Figure 5.4.

   (f) Transactions from T15 to T19 share some prefix with the first branch and hence will lead to the tree shown in Figure 5.5.

   (g) Since transaction T20 had no frequent items it will not be considered in any further calculations.

FP-Tree for sequential patterns

The reordering of the items before insertion in the tree allowed for a more condensed structure hence more efficient mining. However, if the order of the items in the pattern has its own significance, then the reordering is not possible. Even though the same mining mechanism can produce the correct results, the processing would not be as fast if the items were re-ordered.

Using the same database in the example above, assuming that the order of the items is of significance, after eliminating the infrequent items, the transactions to be inserted in the tree will be as shown in Table 5.4. By comparing the FP-tree built while keeping the order of the items (Figure 5.7) to the FP-tree built before re-ordering the items (Figure 5.5), we can see that it is not as condensed.

5.3.2 Applicability to Sequential Patterns

The mechanism explained in [26] describes mining the FP-tree of itemset patterns; we suggest that it is applicable for mining the sequential patterns as well. For both the sequential pattern tree and the item-set pattern tree, the FP-Growth mining process builds the frequent patterns recursively starting with the frequent single items identified during the construction of the tree. For each single item, all the prefix patterns of that item are considered as a conditional database, based on which a conditional FP-tree is constructed.

As mentioned before, the first step in constructing the conditional FP-tree is to count the frequency of each of the items in the conditional database (i.e. those that form a 2-item pattern with the item under consideration). All items that do not meet or exceed the minimal threshold are eliminated, then the new tree is built based on the frequent items. The conditional tree is then mined the same way until we reach a conditional tree that contains only one branch. This tree does not need to be mined; all the combinations of items in that branch are considered frequent patterns.

5.4 FP-Growth mechanism for the associated sequence mining problem

We now explain the extension of the FP-growth mechanism to be used for the associated sequence problem.

Since we know beforehand which parts of the transaction should be in the antecedent and which part will be in the consequent, we can use this knowledge to add extra pruning. That is, any association rule must meet or exceed a minimum confidence threshold in order to be considered. Since the support of the pattern $(P_1 \bigcup P_2)$ in the entire database is equal to the support of P_2 in the conditional pattern base of P_1, then according to equation 5.1, for any pattern P_c to appear as a consequent of a rule with a pattern P_a as the antecedent, the support of P_c in the conditional pattern base of P_a should equal or exceed the support of P_a in the entire dataset. In other words, the minimum support threshold of the sequential patterns should be raised according to equation 5.2:

$$\xi_{P_c} = support(P_a) * \beta \tag{5.2}$$

where ξ_{P_c} is the support of P_c in the conditional pattern base of P_a

Hence we need to keep the sequential patterns in separate sub-trees in the tail of the item-set patterns. This separation would also allow us to separate the processing and limit the over-head to keeping the order of the sequential patterns only when it is needed.

5.4.1 The construction of the mixed FP-tree

We need a data structure that distinguishes between the itemset part and the sequenceset part of each record; we call this the mixed tree. The construction of the mixed tree is similar to what was described before, except that the last element of the itemset part of a transaction will refer to a separate "tail tree" where the sequential elements are organized in a sequential FP-tree.

All prefix elements in the path that contain that element should refer to the tail tree as well as to any tail tree of any intermediate elements. When adding a new transaction, the sequential part should be inserted into the tail tree attached to the final element in the path, if it exists; otherwise a new tail tree should be created.

For example, the mixed tree for the dataset in Table 5.5 should look like the tree shown in Figure 5.8 for a minimum support equal to 3.

The algorithm for constructing the mixed FP-tree is provided in the listing below:

Algorithm 1: Construction of the mixed-FP Tree

Input:

dataset of the format < *itemset*#*sequenceset* >

minimum support

Output:

A mixed FP-Tree

1. First scan for the dataset
   (a) Count the frequency of each single element (in both the itemset and sequence-set parts)
   (b) Eliminate single elements that do not satisfy minimum support threshold
   (c) Sort the itemset elements according to their frequency
2. Second scan for the dataset: For each transaction:
   (a) Eliminate elements that do not satisfy minimum support threshold from the transaction
   (b) Re-order the itemset part according to the frequency of the items
   (c) Insert the transaction into the tree

T1,b,g,i,h,j,c,f,d,a
T2,c,d,f,e,a,g,b,h
T3,f,j
T4,a,b,h,j,d,i,e,g,f
T5,f
T6,i,g,b,e,a,d,f,c
T7,e,i,d,h,f
T8,g,f,b
T9,d,h,a,i,f
T10,b,a,c,f,d,g,i,e,h,j
T11,f
T12,e,b,g,a,h,c,i
T13,i,c,h
T14,e,b,h,g,c,i,j
T15,f,d,c,e,j,b
T16,c,f,h,e,g,a,i,j,b,d
T17,b,c,f,h,j,a,i,g,e
T18,f,d,h
T19,j,f,d
T20,j

Table 5.2: Example Dataset

i. Starting at the root of the tree as the current node; if there is a child of the current node having the same element name, then increment its count,

ii. otherwise, create a new node for that element and set its count to one

iii. Set that new node to be the current node and add the rest of the transaction to it

iv. Once the "#" delimiter is reached

   A. If the last node has a tail sequence tree, insert the sequence set to that tree; Otherwise

   B. create a new sequence tree as the tail tree of that node

   C. insert the sequence set to the new tree

   D. for every parent of the current node add a pointer to that tail tree

T1,f, h, i, d, b
T2,f, h, d, b
T3,f
T4,f, h, i, d, b
T5,f
T6,f, i, d, b
T7,f, h, i, d
T8,f, b
T9,f, h, i, d
T10,f, h, i, d, b
T11,f
T12,h, i, b
T13,h, i
T14,h, i, b
T15,f, d, b
T16,f, h, i, d, b
T17,f, h, i, b
T18,f, h, d
T19,f, d

Table 5.3: Example dataset after infrequent items removed and items reordered

T1,b, i, h, f, d
T2,d, f, b, h
T3,f
T4,b, h, d, i, f
T5,f
T6,i, b, d, f
T7,i, d, h, f
T8,f, b
T9,d, h, i, f
T10,b, f, d, i, h
T11,f
T12,b, h, i
T13,i, h
T14,b, h, i
T15,f, d, b
T16,f, h, i, b, d
T17,b, f, h, i
T18,f, d, h
T19,f, d

Table 5.4: Example dataset after infrequent items removed; the order of items is kept as is

T1,g,c#S2,S8,S1,S4,S3,S7,S5,S6,S9
T2,g,c,b,i,a,d,e,f#S3,S1
T3,f,h,e#S7,S4,S3,S5,S1,S9,S8,S6,S2
T4,f,b,h#S3,S1,S7,S6,S2,S9
T5,f,c,g,h,d,i,b,a,e#S5,S2,S4
T6,i,b,f#S3,S5
T7,b#S6,S7,S4
T8,a,c,h#S8,S5,S1,S4,S7,S3,S2,S9
T9,c,b,g,d,h,i,e,a#S1,S9,S4,S2,S3,S6,S5
T10,b,i,e,a,c,h,d,f,g#S1,S3,S9,S8,S4,S5,S6

Table 5.5: Mixed Dataset

5.4.2 FP-growth to mine the mixed FP-tree

Again the FP-growth mining process is similar to the description provided in section 5.3.2. However, it requires an extra processing step before building the conditional tree, which is to merge all the sequential trees that the item under study is referring to, and mine this merged tree to find out the sequential patterns that satisfy the minimum confidence of forming a rule with the itemset. The algorithm of mining the mixed FP-tree is provided in the listing below:

Algorithm 2: FP-Growth to mine the mixed FP-Tree

Input:

Mixed FP-Tree

Minimum support

Minimum confidence

Output:

Association rules of the format itemset → sequenceset

1. If the tree has one branch only then output each combination of the elements in the branch as the frequent set

2. otherwise, for each element in the frequent item-set (starting with the least frequent),

build conditional pattern base of that element

(a) For each node in the tree that contains that element

i. Find the prefix elements (i.e. parent nodes)

ii. The support of this item-set equals the count of the node under consideration

iii. Dissolve the sequence tree set as the tail tree of the current node as well as all sequence trees that the current node has a pointer to; so that, each branch represents a sequence-set with the support equal to the count of its leaf node

iv. Assign the item-set from (2-a-i) and the set of sequence-sets from (2-a-iii) to represent a transaction in the conditional pattern base in the format $< itemset\#sequencesets >$

(b) Build a sequence tree of all the sequence-sets found in (2-a-iii)

(c) Calculate the new minimum support for the antecedents that satisfy the minimum confidence

(d) Mine the sequence tree based on the new calculated minimum support

(e) Build a conditional mixed-tree out of the conditional pattern tree

(f) Recursively mine the new tree

5.5 Performance Study

The results obtained by the mixed FP-growth algorithm could have been achieved by just treating the entire transaction as a sequence based transaction. However, this would have affected the performance very severely, as experimental results described in this section show.

5.5.1 Design

In order to evaluate the benefit of having the mixed structure, two experiments were performed. The first experiment (referred to as experiment 1) aimed at evaluating the cost in performance

T1,h,g,e,b,a,c,d,i
T2,i,b,c,e,a,g,f,h
T3,c,b,a,g,d,f,i,h
T4,d,i,c,e,b
T5,b,a,g,i,h,d
T6,g,a
T7,d,e
T8,a,d,e,h,i,g
T9,b,i,f,c
T10,c,b,f,a,d,g,i,h

Table 5.6: Examples of randomly generated records

caused by maintaining the sequence of the elements. The second experiment (referred to as experiment 2) aimed at evaluating the benefit of using the mixed structure.

To achieve that goal of the first experiment, a pool of 100 datasets was generated randomly. The algorithm below describes the generation of one dataset drawn from the 10-element set {a,b,c,d,e,f,g,h,i,j}.

1. For records 1 to 1000

   (a) Generate a random number *n* between 1 and 10 to represent the length of the record

   (b) Generate *n* distinct random numbers between 1 and 10 to indicate the chosen elements. (The selected numbers are kept in a set. Whenever a new random number is generated it is checked whether it has been selected before. If it was, then a new random number is generated until a distinct number is found then added to the set)

   (c) Print the elements corresponding to the generated *n* random numbers as one record in the dataset

Table 5.6 gives an example of 10 records that can be generated using the illustrated algorithm.

For each of the generated datasets, two FP-Growth trees were built and mined; once with preserving the order of the items, that is, considering it as a sequenceset, and once with ne-

T1,f,e,d,c,b,a,h,g,i#S7,S6,S5,S1,S4
T2,i,h,g,d,e,b,a,c,f#S7,S9,S1,S8,S6,S5,S4
T3,a#S1,S3,S6
T4,f,g,h,c,e,i,d#S6,S8,S4,S1,S3,S2,S7,S9,S5
T5,h,d,f#S4,S1,S7
T6,h,g,b,i,e,d,c,f,a#S1,S3
T7,h,a,g,i,b,e#S1,S4,S2,S8,S3
T8,h,e,c#S6,S4,S9,S1,S5,S3,S7,S2,S8
T9,e,a,f,d#S5,S7,S9,S4
T10,a,e,b,c,f,h,g,d,i#S3,S1,S2,S9,S7,S6

Table 5.7: Examples of randomly generated records

glecting the order of the items, that is, considering it as an itemset. The time for building and mining each of the trees was recorded.

To achieve the goal from the second experiment, another pool of 1000 datasets was generated using the same algorithm described above, except that for each record two parts were generated. One part was drawn from the set {a,b,c,d,e,f,g,h,i,j} and the other part was drawn from the set {S1,S2,S3,S4,S5,S6,S7,S8,S9,S10}.

Table 5.7 gives an example of 10 records that could be generated for the second experiment.

In these experiments, three different FP-Growth trees were built for each dataset. One tree was built with preserving the order of the items, that is, treating the dataset as a sequenceset; another tree was built with neglecting the the order of the items completely, that is treating the dataset as an itemset; and the third tree was a mixed tree, that is, the order of each record was neglected for the first part but preserved for the second part. Again, times for building and mining each of the trees were recorded.

5.5.2 Results

Figure 5.9 shows a boxplot comparison between the time cost of building the FP tree when the datasets are treated as itemsets and the time cost of building the FP tree when the datasets

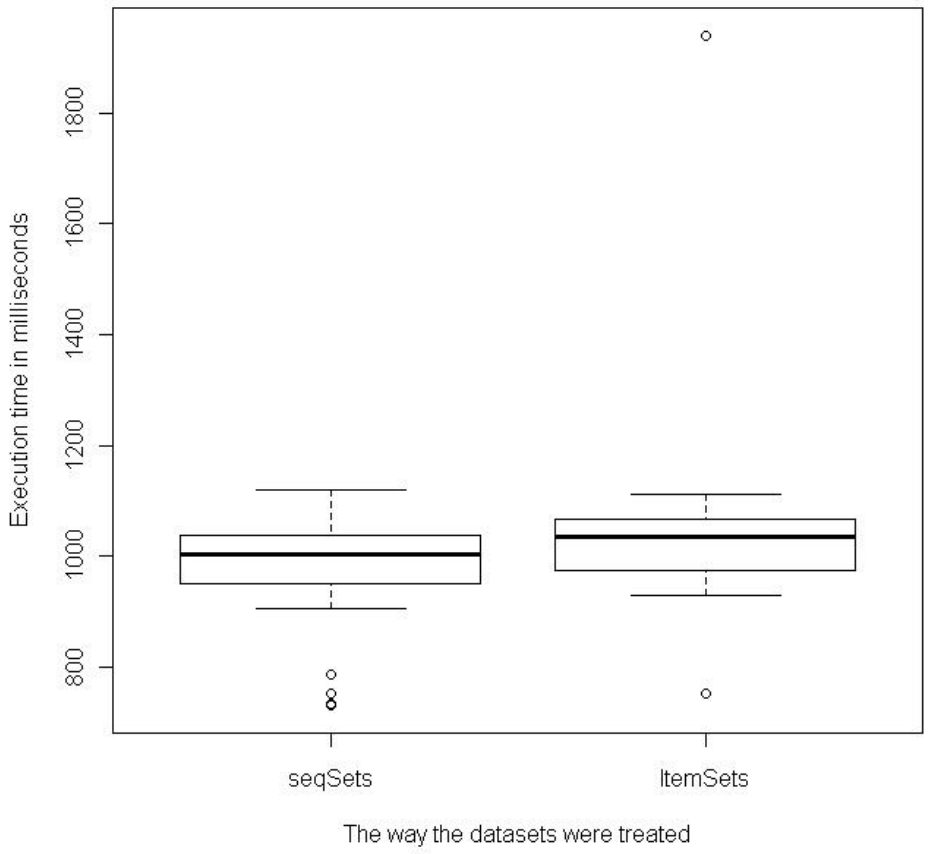

Figure 5.9: Time taken to build the FP trees in experiment 1

were treated as sequence sets[2]. The cost of building the tree in both cases was on average very similar.

On the other hand, the cost of *mining* the sequence based trees was significantly higher than that of mining item based trees, as shown in Figure 5.10.

The overall performance of both building and mining the trees was affected by this difference, making the consideration of the order relationships between the elements costly, as shown in Figure 5.11. On average, it took slightly more than 17% longer to build and mine

[2]In each boxplot, the heavy line shows the median, and the top and bottom of the box show all the points except the top 25% and bottom 25%. The distance from the top to the bottom of the box is the interquartile distance. All the points greater than 1.5 times the interquartile distance from the top or bottom of the box are outliers, shown as open circles. The "whiskers" above and below the box extend to the maximum and minimum non-outlier points.

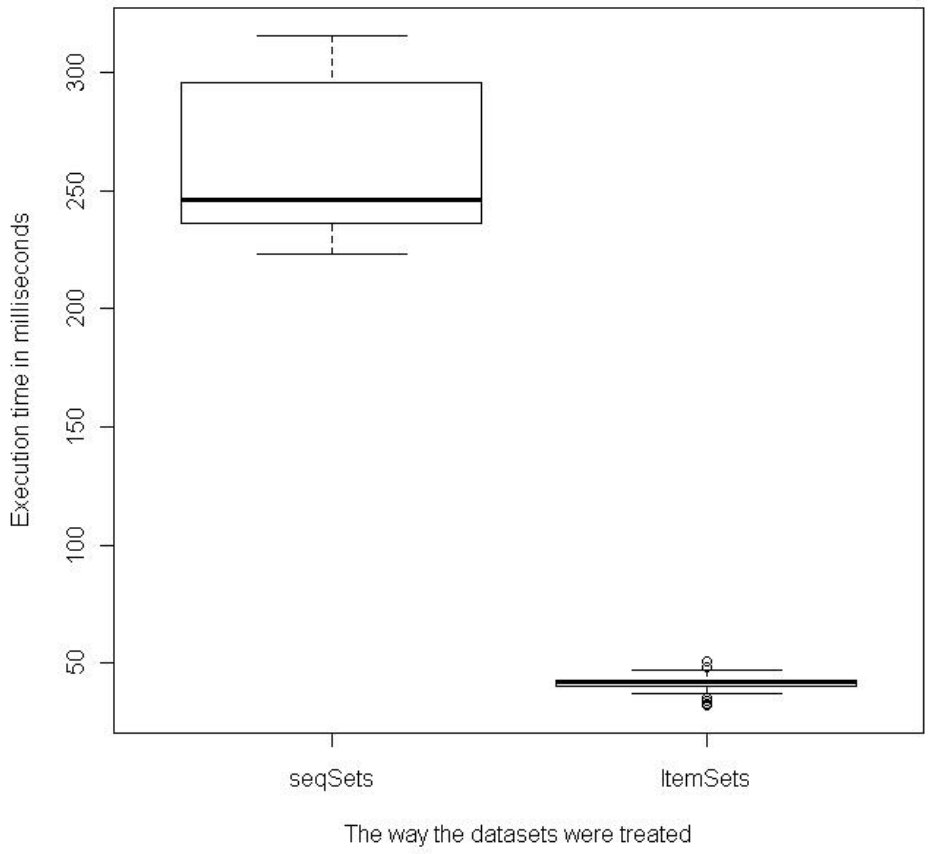

Figure 5.10: Time taken for mining the FP trees in experiment 1

the FP-trees when order was considered.

Figure 5.12 shows the time consumed for building the tree when the datasets were treated as mixed sets, compared to the time consumed when the datasets were treated as itemsets and when they were treated as sequence sets. Obviously, building the mixed tree consumes more time.

However, as shown in Figure 5.13, mining the mixed tree is more efficient than treating the entire dataset as a sequence based one. The mining of only 25 sequence based datasets were completed, while the remaining 75 datasets produced a stack-overflow error. In addition to that, the mining of the mixed tree produces the association rules directly, while the mining of the

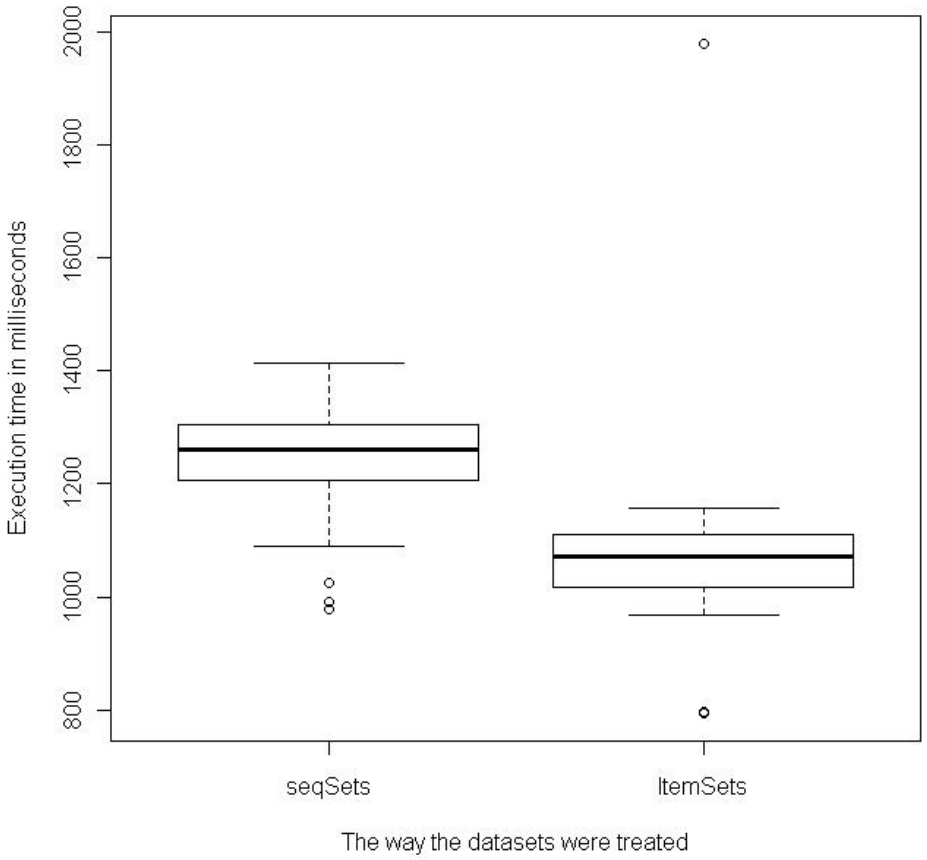

Figure 5.11: Overall time taken for buliding and mining the FP trees in experiment 1

sequence or item based trees only generated only frequent itemsets; generating the association rules would need additional computations.

The overall time needed for building and mining the mixed tree is less than that needed for the sequence based tree. Even though it is significantly more than the time needed for building and mining an item based tree, this cost is unavoidable, since for some parts of the records the order relationship has to be maintained. The comparison between the overall times needed for building and mining the trees is shown in Figure 5.14.

The results of the comparisons showed that using the mixed-structure enhances the performance significantly.

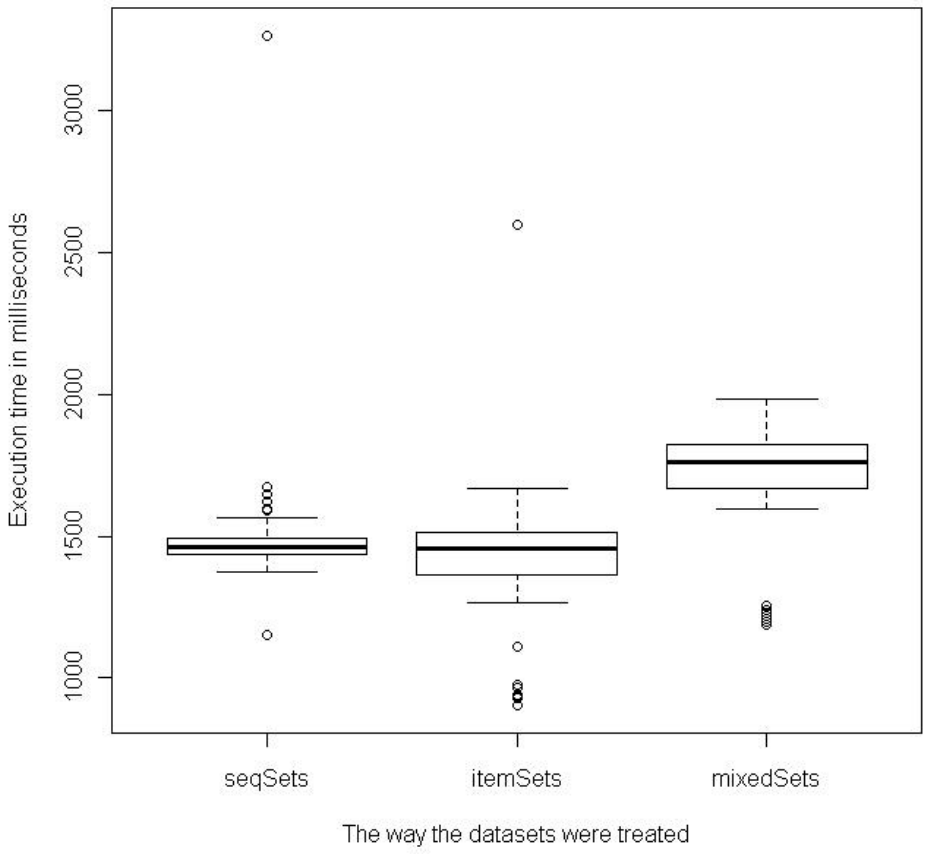

Figure 5.12: Time taken to build the FP trees in experiment 2

5.6 Accuracy Study

In the accuracy study we evaluate the results of the localization process and the usability of the associated sequence mining in comparison to a variation of two existing techniques, namely, Tarantula [32] and Ochiai [2]. Because we produce results at the class level, for this experiment we adapted Tarantula and Ochiai to produce results at class level in order to be able to compare them to our results.

In section 5.6.1, we describe the subject programs and the faults we used, analyzing the types of faults according to our classification from chapter 3. In section 5.6.2, we describe how we collected the data. In section 5.6.3, we discuss the results of the study.

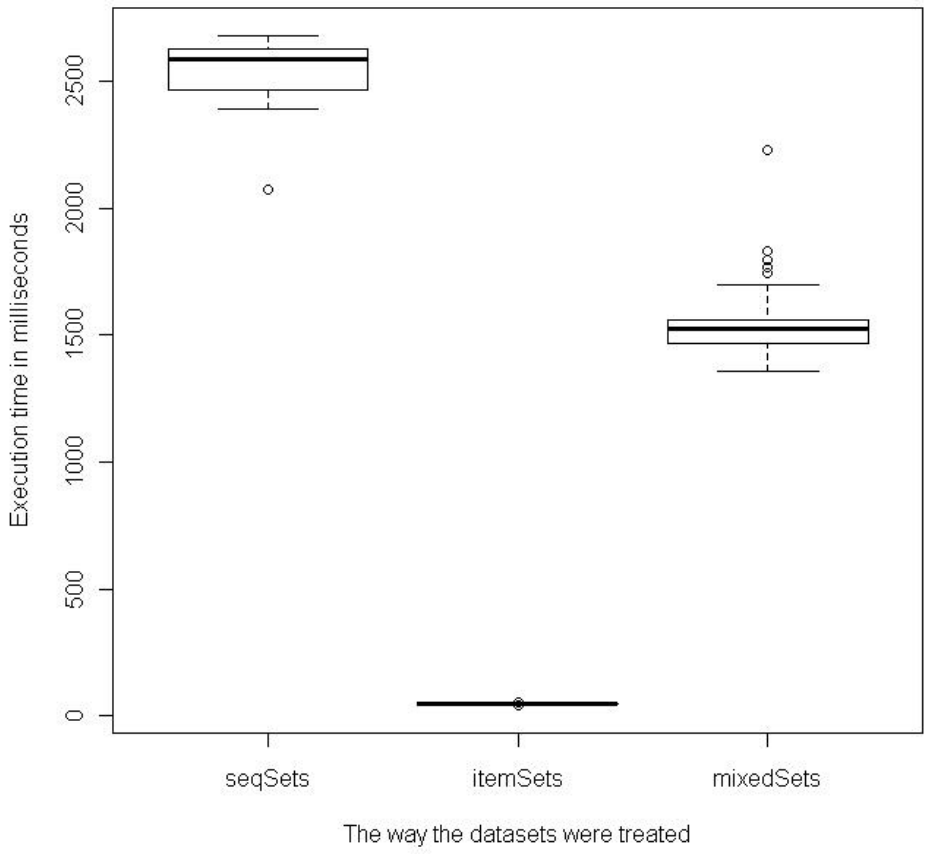

Figure 5.13: Time taken for mining the FP trees in experiment 2

5.6.1 Subject Programs and Faults

Table 5.8 shows the subject programs we have used to evaluate the localization process. These are three different versions of JTopas [3], a Java library for parsing text data, three different versions of XML-security [4] , a component library that implements XML signature and encryption standards, and two different versions of JMeter[5], a desktop application for functional and performance testing. The number of classes range from 19 classes in version 1 of JTopas to 369 classes in version 5 of JMeter. All the subject programs and their test suites were downloaded

[3]http://jtopas.sourceforge.net/jtopas
[4]http://xml.apache.org/security
[5]http://jakarta.apache.org/jmeter

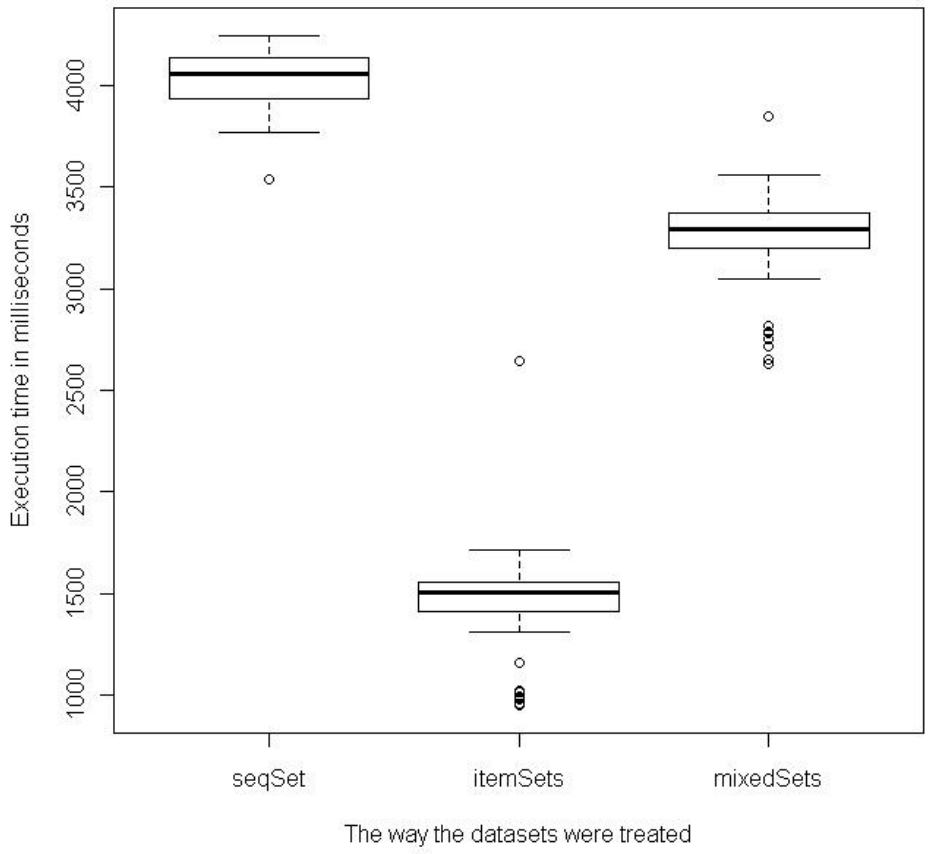

Figure 5.14: Overall all time taken for building and mining the FP trees in experiment 2

from the Software Infrastructure Repository (SIR) [6]. The repository also provides a number of faults that can be seeded to the different subject programs; however, not all of them yielded failures.

Tables 5.9 and 5.10 show the names, locations and types of the seeded faults that yielded failures in the 6 different versions of jtopas and xml-security. For jmeter, only two of the downloaded faults yielded failures; therefore we seeded new faults to provide diverse types and locations of the faults for the sake of better evaluation of the techniques. The names, types and locations of the faults of jmeter are shown in Table 5.11.

[6] http://sir.unl.edu/content/sir.php

Subject Program	Number of Classes	Number of Faults	Number of test cases
jtopas V1	19	5	196
jtopas V2	21	3	130
jtopas V3	50	5	183
xml-Security V1	217	7	92
xml-Security V2	218	5	94
xml-Security V3	146	4	84
jmeter V4	366	10	74
jmeter V5	369	11	83

Table 5.8: Subject Programs

Version	Fault	The faulty class	Fault type
V1	Fault_2	ExtIOException.	1.2.2.2 Object State Fault
	Fault_3	ExtIOException	1.1.1.1.2.1 Missing overload fault
	Fault_5	AbstractTokenizer	2.2.1.1.2.2 Faulty value(s)
	Fault_9	ExtIndexOutOfBoundsException	1.1.1.1.2.1 Missing overload fault
	Fault_10	ExtIndexOutOfBoundsException	2.1.1.1. Logical Condition fault
V2	Fault_1	ExtIOException	2.2.3.1 Missing call fault
	Fault_3	ThrowableMessageFormatter	2.1.1.1. Logical condition fault
	Fault_7	AbstractTokenizer	2.1.1.1. Logical condition fault
V3	Fault_4	PluginTokenizer	2.2.3.1 Missing call fault
	Fault_6	StandardTokenizer	2.1.2.2. Local operation fault
	Fault_7	StandardTokenizer	2.2.1.2.1 Faulty return value
	Fault_8	StandardTokenizer	2.2.1.4 Wrong message sent
	Fault_9	StandardTokenizer	2.2.1.4 Wrong message sent

Table 5.9: jtopas faults' types and locations

In each of tables 5.9, 5.10 and 5.11, the classification of the faults is taken from the fault taxonomy given in chapter 3. We also provided a numbered listing of the taxonomy in appendix A.

5.6.2 Data Collection

We collected data from the previous experiments aiming towards answering three main questions:

1. Is it sufficient to collect behavior information in association of the objects' variable

Version	Fault	The faulty class	Fault type
V1	CE_HD_2	Canonicalizer20010315Excl	2.1.1.1 Logical condition fault
	CE_HD_3	Canonicalizer20010315Excl	2.1.1.1 Logical condition fault
	CEWC_HD_1	Canonicalizer20010315Excl-WithComments	2.2.1.1.2.2. Faulty value(s)
	CHP_AK_1	C14nHelper	2.1.1.1 Logical condition fault
	CN2_AK_2	Canonicalizer20010315	2.1.1.1 Logical condition fault
	CNC_AK_1	Canonicalizer	1.2.3 Class state fault
	XSI_AK_1	XMLSignatureInput	2.2.3.1 Missing call fault
V2	CH_HD_1	C14nHelper	2.1.1.4.1. Missing branch fault
	CHP_AK_1	C14nHelper	2.2.1.1.2.2. Faulty values(s)
	RF_HD_2	Reference	2.2.3.1. Missing call fault
	C2E_AK_1	Canonicalizer20010315Excl	2.2.1.1.2.2. Faulty values(s)
	EP_AK_1	ElementProxy	2.1.2.2 Local. operation fault
V3	RF_HD_1	Reference	2.2.1.1.2.2. Faulty values(s)
	XU_HD_1	XMLUtils	2.2.1.4. Wrong Message sent
	FAULT_ADD_1	XMLSignatureInput	2.1.1.2.1. Early branching fault
	FAULT_ADD_2	ResourceResolverSpi	2.1.1.1. Logical condition fault

Table 5.10: xml-security faults' types and locations

states?

2. How much reduction in the search space can each of the techniques provide?

3. Which of the localization techniques provided better ranking?

We will deal with each of these questions in turn.

(1) Is it sufficient to collect behavior information in association of the objects' variable states? That is, if a class does not have any member variable defined, should we consider the different behaviors exhibited by its methods or not? In order to answer this question we had two different behavior log files for each seeded fault, one containing the behavior of objects that did not have any member variable (we call this constant state), and the other only containing behavior lines associated with member variables.

(2) How much reduction in the search space can each of the techniques provide? We compare the failing behavior to the successful behavior, with the objective of eliminating behavior that is not likely to cause the failure. We do this in order to reduce the amount of code that the

Version	Fault	The faulty class	Fault type
V4	FAULT_101	GenericController	2.1.1.1 Logical condition fault
	FAULT_102	FunctionProperty	2.1.1.1 Logical condition fault
	FAULT_103	ProxyControl	2.1.1.1 Logical condition fault
	FAULT_104	ProxyControl	2.1.1.2.1. Early brancing fault
	FAULT_105	ProxyControl	2.1.1.2.2. Late branching fault
	FAULT_106	ProxyControl	2.1.1.3.2. Late merging fault
	FAULT_107	ProxyControl	2.1.1.4.1. Missing branch fault
	FAULT_108	FunctionProperty	2.1.1.4.2. Extra branch fault
	FAULT_109	CookieManager	2.1.1.1 Logical condition fault
	FAULT_110	CookieManager	2.2.3.1. Missing call fault
V5	FAULT_101	Argument	2.1.2.2. Local operation fault
	FAULT_102	CompoundVariable	2.1.1.1. Logical condition fault
	FAULT_103	Load	2.2.1.1.2.2. Faulty value(s)
	FAULT_104	Save	2.2.3.1. Missing call fault
	FAULT_105	Save	2.1.1.2.2. Late branching fault
	FAULT_106	ListedHashTree	2.2.1.1.2.1. Switched arguments fault
	FAULT_107	TestCompiler	2.2.1.4. Wrong message sent
	FAULT_108	StringUtilities	2.2.3.3. Call order fault
	FAULT_109	HTTPSampler	1.1.1.1.2.1. Missing overload fault
	FAULT_110	JMeterContext	2.2.3.3. Call order fault
	FAULT_111	RegexExtractor	2.1.2.1. Local operation fault

Table 5.11: jmeter faults' types and locations

programmer/debugger will have to examine, based on the recommendation of the localization mechanism.

For the state-dependent process, we made the claim that eliminating every behavior line that occurred during a successful test case would be confusing as a failing behavior may occur during successful test cases as well. In order to validate this claim, we highlighted suspicious classes in three different ways, once after eliminating every successful behavior (we call this comparison with raw success), then classes after eliminating behaviors identified as success rules by the associated mining algorithm, and finally, without eliminating any behavior in comparison to the success. For Tarantula and Ochiai, even though the original technique did not aim to eliminate any part of the code completely, we considered parts of code that got a suspiciousness of zero as being eliminated by the technique in order to evaluate our proposed mechanism in comparison to them.

(3) Which of the localization techniques provided better ranking? Since our proposed mechanism produces rankings at the class level, while both Tarantula and Ochiai provide ranking at the line level, we had to transform the line suspiciousness to a class suspiciousness, by considering the suspiciousness of the class to be the maximum suspiciousness of its lines.

5.6.3 Results

Figure 5.15 is a bar chart that shows a comparison between the numbers of faults localized by each of the techniques both when the constant state behavior was considered and when it was eliminated. The bar chart shows that all the 50 faults were localized when objects with constant states were taken into account, except when comparison was performed against the raw successes, i.e. when all behavior lines that appeared in the success log were eliminated from the failing log. In particular, 12 faults (24%) were missed due to the elimination of constant state behavior and additional 12 (24%) were missed due to comparison with raw success.

Figures 5.16 through 5.24 show boxplots to compare the sizes of suspicious class lists

(in percentage with respect to the actual size of the respective subject program) and the ranks (shown as the percentage of the number of classes needed to be considered before the programmer reaches the actual faulty class). Note that in these box plots, "smaller is better" because smaller values show less information to be processed by the user. We start with the box plots drawn for the all of the observations for the two measures both when constant state behavior was considered and when it was eliminated (2 x 2 x 50 = 200). Then we provide the box plots for some of the subject programs individually.

We can observe from comparing the figures that eliminating the constant state behavior seems to enhance the overall ranking and sizes of the suspicious lists. The same enhancement is provided when comparison to raw success was performed.

The comparison to success rules generated by the associated sequence mining algorithm did not seem to provide any improvement on the ranking than when the comparison step was skipped altogether. Comparison to raw success seemed to provide the best ranking, even better when constant state behavior was not considered, followed by Ochiai and then Tarantula, followed by the state-dependent ranking after comparison with success rules and when skipping the comparison step.

Again, the comparison with raw success provided the smaller suspicious list, even smaller when eliminating constant state behavior, followed by the comparison to success rules, then skipping the comparison, then followed by both Tarantula and Ochiai.

Figure 5.20 provides an example of a typical comparison between the sizes of the suspicious lists produced by the different techniques. In Figure 5.22 the improvement of the sizes is more obvious. Figure 5.23 shows an obvious example that Tarantula and Ochiai gave better ranks, while Figure 5.21 shows one of the cases where the state-dependent rankings were actually better than those of both Tarantula and Ochiai. Finally, Figure 5.25 shows an example where the comparison with raw success missed all the faulty classes.

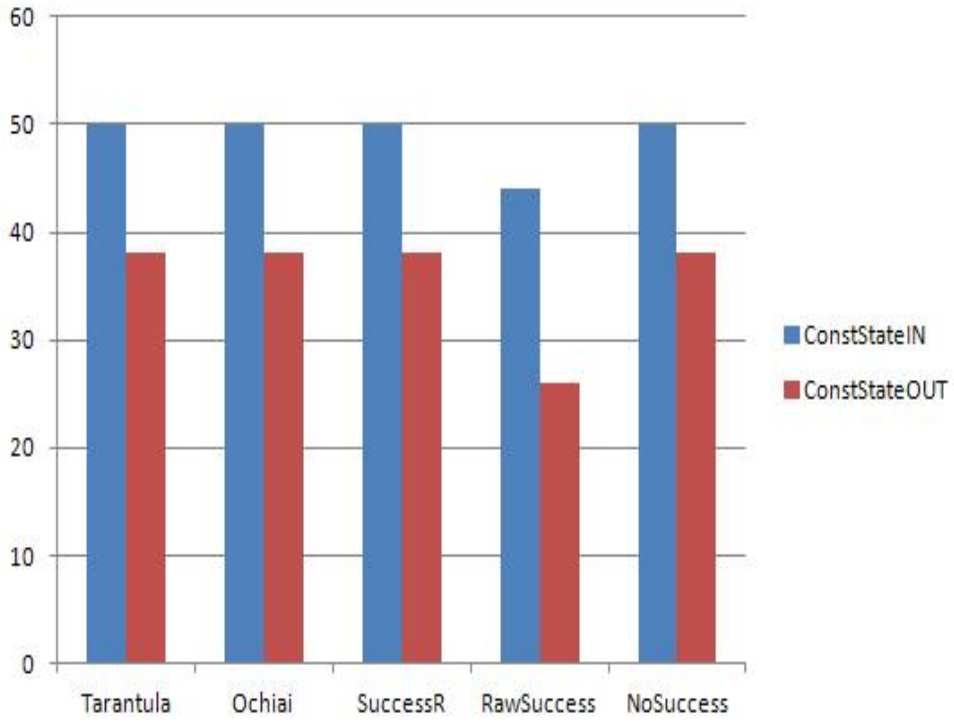

Figure 5.15: Comparison betweeen the number of faults localized

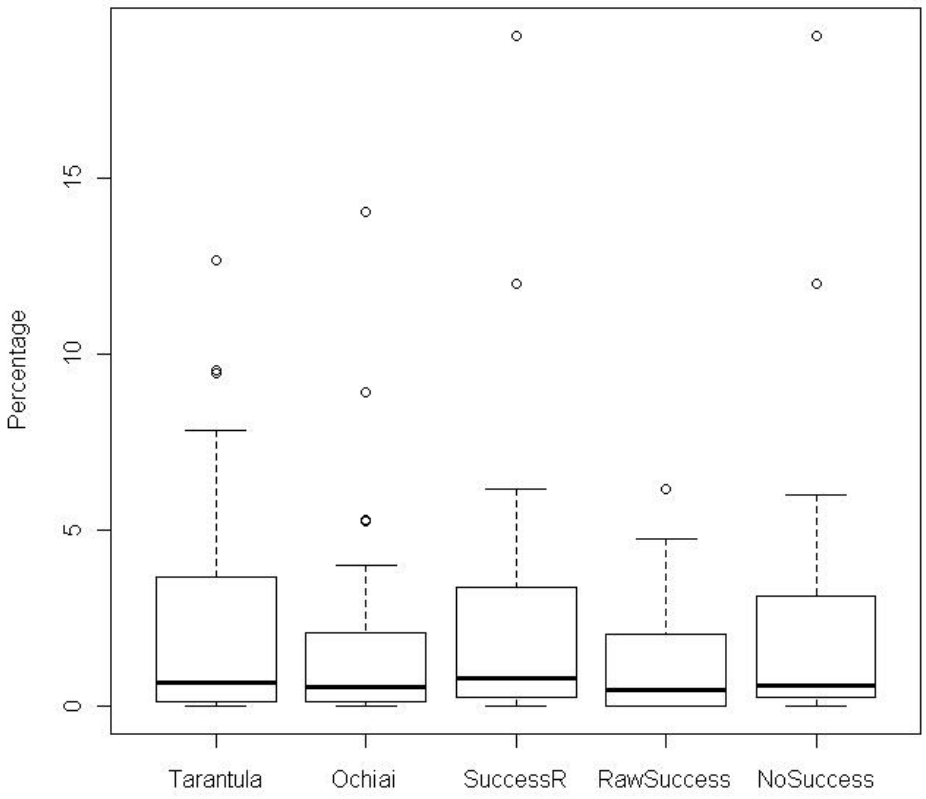

Figure 5.16: Ranks for all seeded faults with constant-state behavior taken into consideration

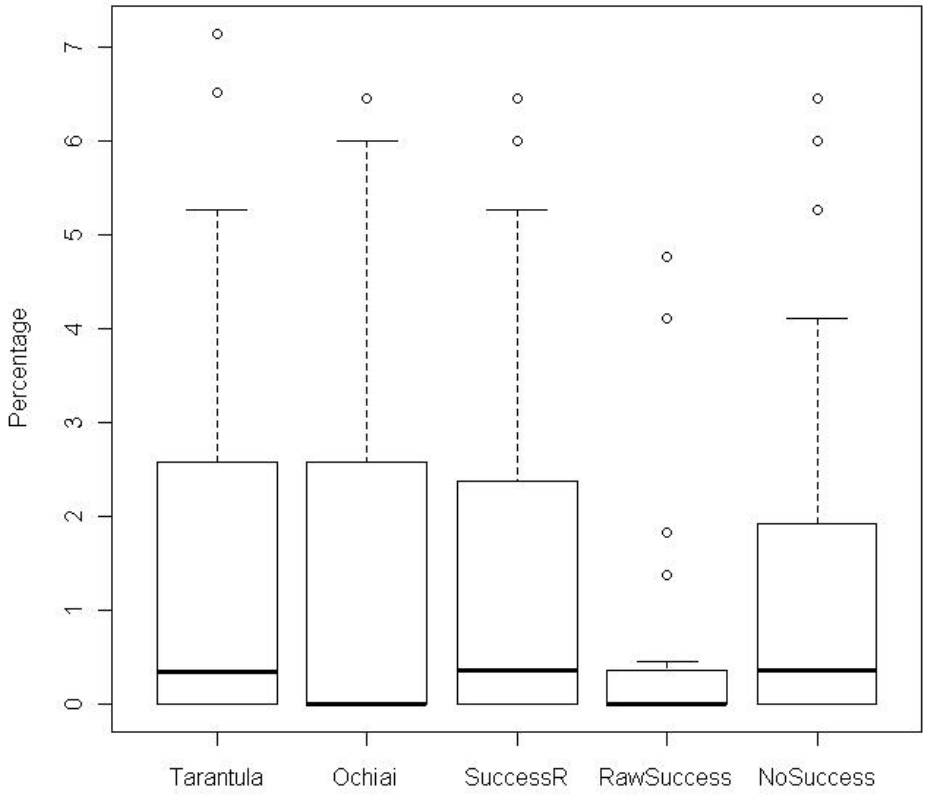

Figure 5.17: Ranks for all seeded faults with constant-state behavior not taken into consideration

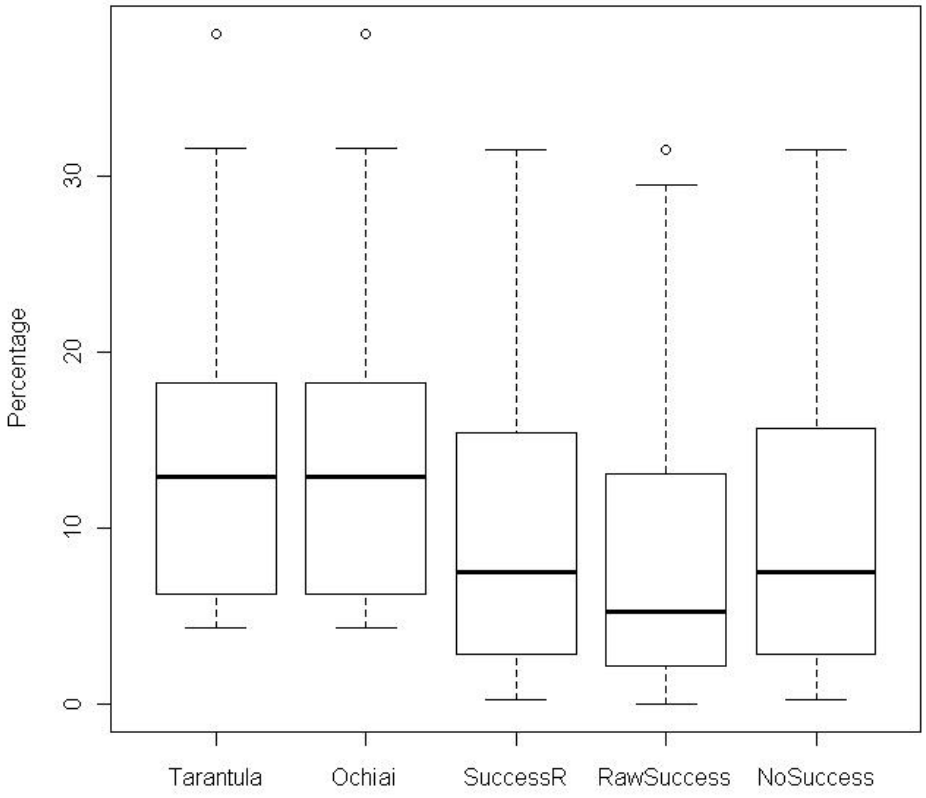

Figure 5.18: Sizes of suspicious lists for all seeded faults with constant-state behavior taken into consideration

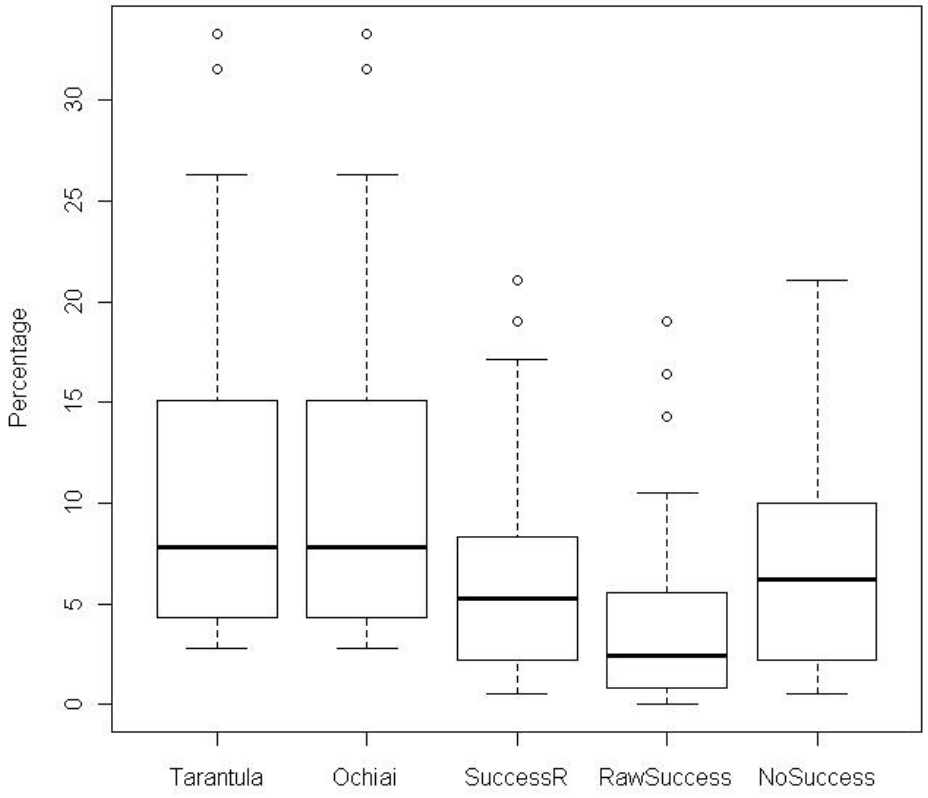

Figure 5.19: Sizes of suspicious lists for all seeded faults with constant-state behavior not taken into consideration

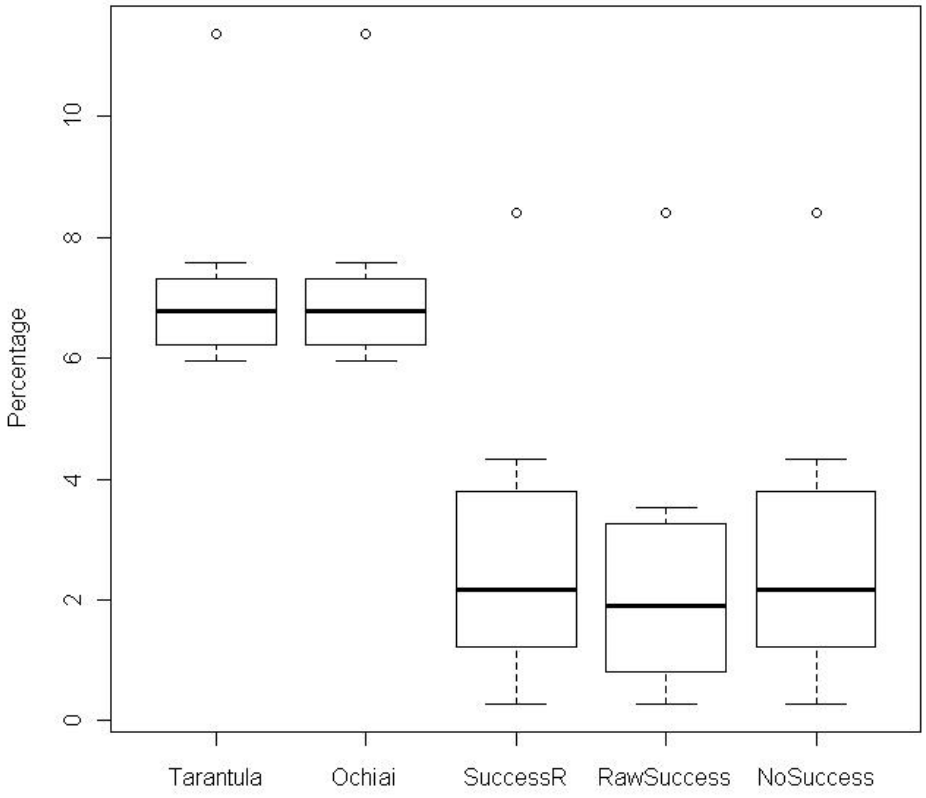

Figure 5.20: Sizes of suspicious lists for JMeterV5 seeded faults with constant-state behavior taken into consideration

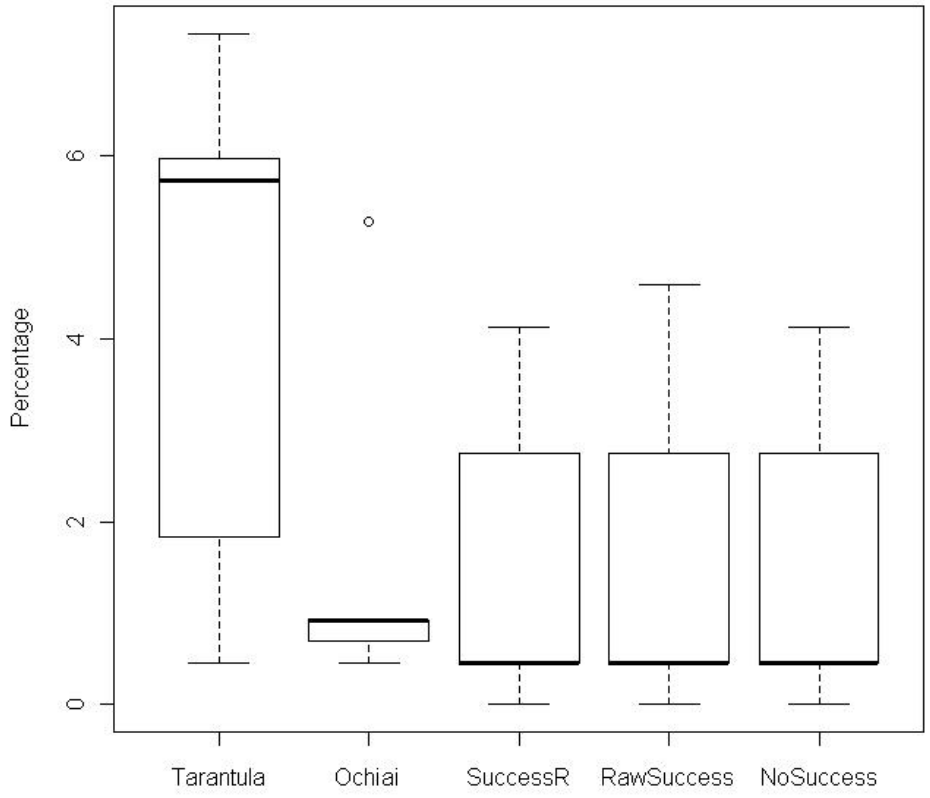

Figure 5.21: Ranks for XMLSecurityV2 seeded faults with constant-state behavior taken into consideration

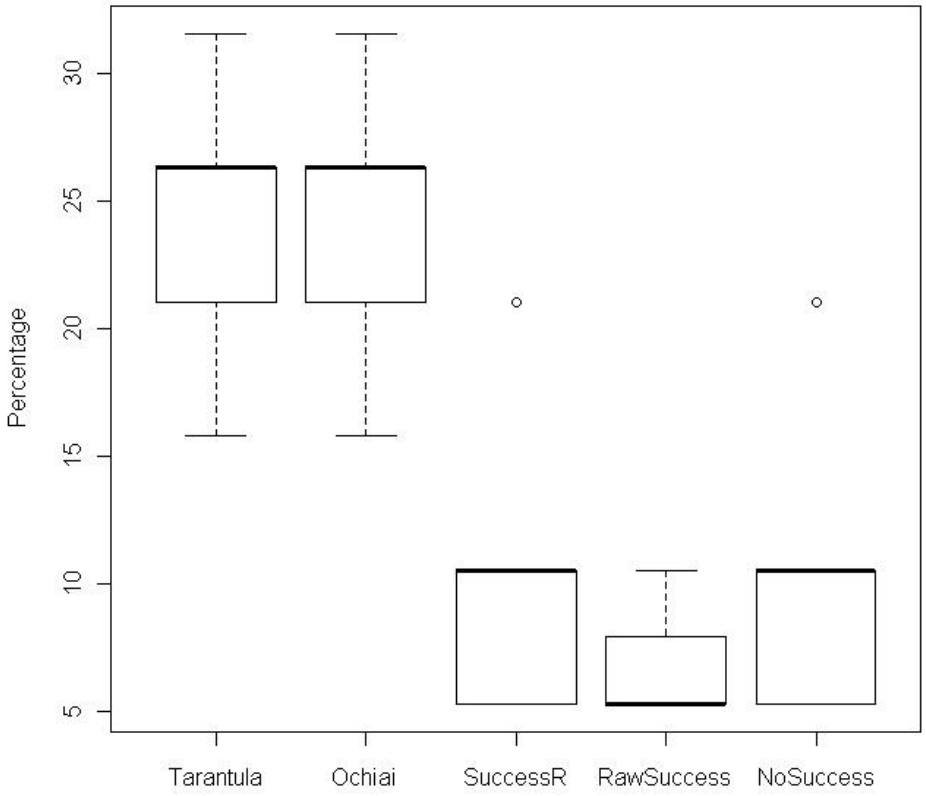

Figure 5.22: Sizes of suspicious lists for JTopasV1 seeded faults with constant-state behavior not taken into consideration

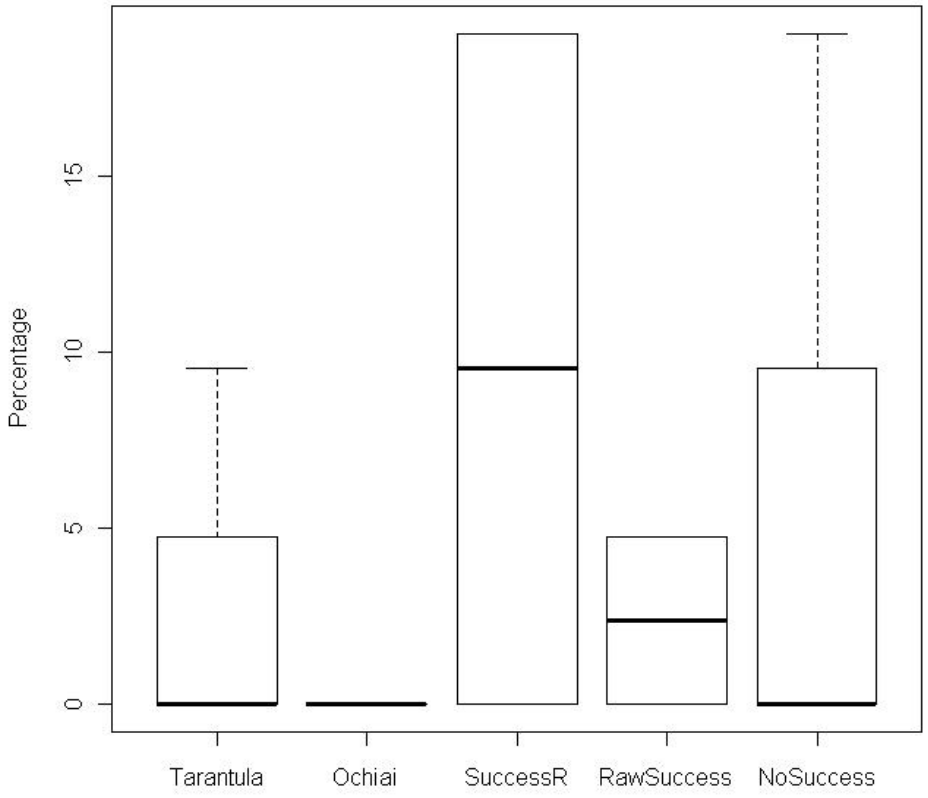

Figure 5.23: Ranks for JTopasV2 seeded faults with constant-state behavior taken into consideration

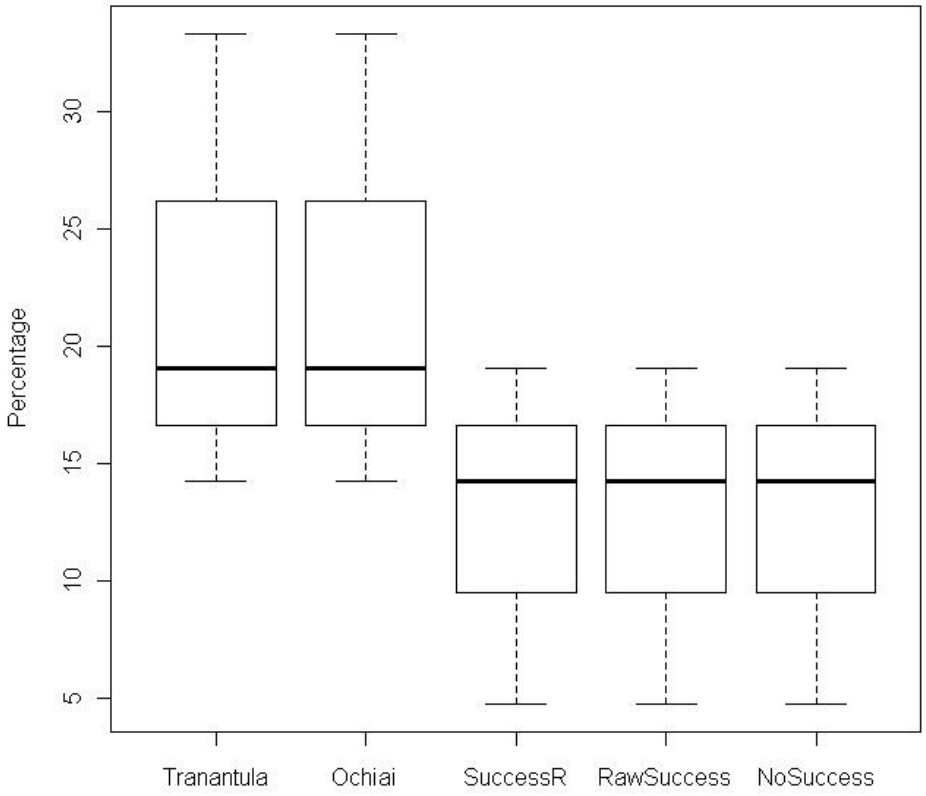

Figure 5.24: Sizes of suspicious lists for JTopasV2 seeded faults with constant-state behavior not taken into consideration

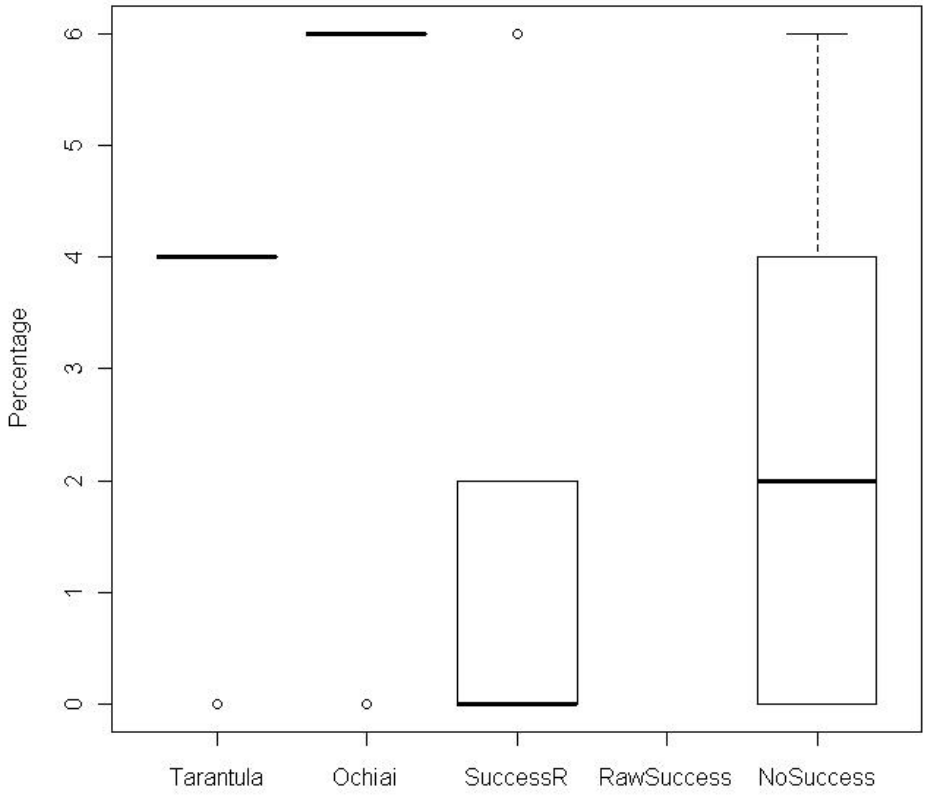

Figure 5.25: Ranks for JTopasV3 seeded faults with constant-state behavior not taken into consideration

5.7 Summary

We can summarize the observations in the following points:

1. Collecting a trace of the state-dependent behavior of the subject program, when taking the constant state behavior into consideration, is sufficient to contain the fault.

2. Eliminating constant state behavior improves both the size of the suspicious lists and the actual fault's ranking, however, there is 24% risk to miss the faulty classes.

3. It seems (from the first few graphs) that raw success is best (slightly) in rank of faulty class, and best by a noticeable amount in sizes of ranked list, however, there is also a risk to miss another 24% of the faulty classes.

4. When extended to give work on the class level, Ochiai and Tarantula provide better rankings in general than those provided by the state-dependent techniques.

5. The state-dependent mechanism provides better sizes of ranked lists.

6. The use of the state associated sequence mining to generate success rules to be used in the comparison step did not provide much enhancement on the size of the suspicious list, however it provided a slight enhancement on the ranking.

Chapter 6

Conclusion

We have explored the usability of classification algorithms in the problem of fault localization. Some of the classifiers did very well on some of the faults, which motivated us to investigate the nature of different faults that may occur in object-oriented software. We created a new UML-based fault taxonomy for implementation faults. We identified around 80 types and subtypes of faults based on the artefacts that appear in the different UML diagrams. More specifically, we focused on the object diagrams and class diagrams of the structured diagrams and sequence diagrams and activity diagrams of the dynamic diagrams. We aimed at making the taxonomy as comprehensive as possible. We had special focus on those faults that would lead to runtime failures.

State-dependent faults seemed to be the best type of fault to be targeted, as it would indirectly allow us to locate other types of faults as well. We created a process that starts by instrumenting the subject program in order to collect only the information that would allow us to investigate the state-dependent behavior of the different objects and classes. We took in consideration both the object state, which is determined by the normal attributes of the class and would differ from object to object, as well as the class state, which is determined by the static members that would only have one value for the class. Throughout the process, we adapted the data mining mindset, that is, we made sure to eliminate whatever data that would not need to be

investigated; hence we eliminated the behavior executed by the teardown and setup methods of the JUnit test suite and considered them as neutral, to restrict the analysis only to the behaviors that we know are either successful or failing.

We developed a customized mining technique, "associated sequence mining", that would work directly on finding out the rules that represent the successful behavior. In fact, we found out that even studying the successful behavior is not as helpful to the localization process, and that it is sufficient to study the failing state-dependent behavior to find the faulty class.

Evaluating the proposed technique, we found out that targeting state-dependent inconsistencies in the behavior of an object oriented program is sufficient to locate the faulty class, even though the original seeded fault was of a different type. In addition, the traces of state-dependent behavior of a subject program are guaranteed to contain the faulty class, only if the constant state behavior is taken in consideration, and it can be used by other localization techniques as well.

6.1 Future Work

For future work, we would like to evaluate our proposed technique on other subject programs with real faults as opposed to the seeded faults. We also would like to evaluate the efficiency of our technique at the method level.

Bibliography

[1] Kjersti Aas, Rangar Bang Husbey, and Mari Thune. Data mining: A survey. Technical Report 942, Norwegian Computing Center, June 1999.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J C Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the accuracy of fault localization techniques. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 76–87, Washington, DC, USA, 2009. IEEE Computer Society.

[5] H.H. Ammar, S.M. Yacoub, and A. Ibrahim. A fault model for fault injection analysis of dynamic UML specifications. In *Proceedings of the 12th International Symposium on Software Reliability Engineering.*, pages 74 – 83, 2001.

[6] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd International Conference on Software Engineering*, volume 1, pages 265–274, 2010.

[7] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. ACM.

[8] Shaimaa Ali Badawy, Ahmed Elragal, and Mahmoud Gabr. Multivariate similarity-based conformity measure (mscm): an outlier detection measure for data mining applications. In *Proceedings of the 26th International Conference on Artificial Intelligence and Applications*, AIA '08, pages 314–320, Anaheim, CA, USA, 2008. ACTA Press.

[9] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, 1983.

[10] Stefan Bruning, Stephan Weissleder, and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, pages 367–368. IEEE Computer, 2007.

[11] Lisa J. Burnell and Eric J. Horvitz. A synthesis of logical and probabilistic reasoning for program understanding and debugging. In *Proceedings of the Ninth international conference on Uncertainty in artificial intelligence*, UAI'93, pages 285–291, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[12] Kai-Yuan Cai. *Software Defect and Operational Profile Modeling*. Kluwer Academic Publishers, 1998.

[13] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In *Proceedings of the 6th international conference on Formal concept analysis*, ICFCA'08, pages 273–288, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] T. Y. Chen and Y. Y. Cheung. On program dicing. *Software Maintenance: Research and Practice*, 9:33–46, 1997.

[15] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *International symposium on software testing and analysis*, pages 141–152, 2009.

[16] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, and Bonnie Ray Man-Yuen Wong. Orthogonal defect classification – a concept for in-progress measurements. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[17] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In *European Conference on Object-Oriented Programming*, pages 528–550, 2005.

[18] Tristan Denmat, Mireille Ducass, and Olivier Ridoux. Data mining and cross-checking of execution traces. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 396–399, 2005.

[19] Trung Dinh-Trong, Sudipto Ghosh, Robert France, Benoit Baudry, and Franck Fleurey. A taxonomy of faults for UML designs. In *Proceedings of 2nd Model Design and Validation (MoDeVa) Workshop*, Montego Bay, Jamaica, 2005.

[20] M. Ducassé and A.-M. Emde. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering*, ICSE '88, pages 162–171, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[21] Mireille Ducasse. A pragmatic survey of automated debugging. In *Proceedings of the first international workshop on Automated and Algorithmic Debugging*, pages 1–15, 1993.

[22] M. P. Gallaher and B. M. Kropp. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute of Standards, RTI-Health, Social, and Economics Research, May 2002.

[23] Georges Gardarin, Philippe Pucheral, and Fei Wu. Bitmap based algorithms for mining association rules. Technical Report 1998/18, University of Versailles, Versailles Cedex, France, http://www.prism.uvsq.fr/rapports/1998, 1998.

[24] Miroslav Grgec and Robert Muzar. Role of UML sequence diagram constructs in object lifecycle concept. *Journal of Information and Organizational Sciences*, 31:63 –74, 2007.

[25] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[26] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12, New York, NY, USA, 2000. ACM.

[27] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of data mining*. MIT Press, Cambridge, MA, USA, 2001.

[28] Jane Huffman Hayes. Testing of object-oriented programming systems (OOPS): A fault-based approach. In *Object-Oriented Methodologies and Systems*, volume 858 of *LNCS*, pages 205–220. Springer-verlag, 1994.

[29] Jane Huffman Hayes. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. In *Proceedings of 14th International Symposium on Software Reliability Engineering*, pages 49–60, 2003.

[30] W. Lewis Johnson and Elliot Soloway. Proust: Knowledge-based program understanding. In *IEEE Transactions on Software Engineering*, volume SE-11, pages 369–380, 1985.

[31] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 273–282, 2005.

[32] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[33] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods, and Algorithms*. John Wiley and Sons, 2003.

[34] Debasish Kundu and Debasis Samanta. A novel approach to generate test cases from UML activity diagrams. *Journal of Object Technology*, 8(3):65–83, 2009.

[35] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, 2005.

[36] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 557–566, 2009.

[37] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. Are faults localizable? In *Proceedings of 9th IEEE Working Conference on Mining Software Repositories*, pages 74–77, 2012.

[38] Leonardo Mariani. A fault taxonomy for component-based software. *Electronic Notes in Theoretical Computer Science*, 82(6):55–65, 2003.

[39] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 128–137, 2008.

[40] Syed Shariyar Murtaza, Mechelle Gittens, Zude Li, and Nazim H. Madhavji. F007: finding rediscovered faults from the field using function-level failed traces of software in the field. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 57–71, Riverton, NJ, USA, 2010. IBM Corp.

[41] Software Engineering Standards Committee of the IEEE Computer Society. IEEE standard classification for software anomalies - IEEE std 1044-1993. *IEEE Standards*, 1993.

[42] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineerin*, pages 84–95, Hong Kong China, 2001. IEEE Computer Society Press.

[43] OMG. Uml superstructure specification version 2.2. February 2009.

[44] Mike Papadakis and Yves Le-Traon. Using mutants to locate "unknown" faults. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 691–700, 2012.

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

[46] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[47] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294, 2009.

[48] Iris Vessey. On program development effort and productivity. *Inf. Manage.*, 10(5):255–266, May 1986.

[49] Dominik Wieland. *Model-Based Debugging of Java Programs Using Dependencies*. PhD thesis, Technische Universitat Wien, 2001.

[50] Zhongxing Yu, Hai Hu, Chenggang Bai, Kai-Yuan Cai, and W.E. Wong. GUI software fault localization using n-gram analysis. In *Proceedings of 13th International Symposium on High-Assurance Systems Engineering*, pages 325 –332, 2011.

[51] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT - Foundations of Software Engineering*, pages 1–10, 2002.

[52] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318, 2009.

Appendix A

Numbered faults of UML-Based taxonomy

1. Structural fault

1.1. Class structure fault

1.1.1. Class definition

1.1.1.1. Inheritance fault

1.1.1.1.1. Depth fault

1.1.1.1.2. Overload fault

1.1.1.1.2.1. Missing overload fault

1.1.1.1.2.2. Extra overload fault

1.1.1.2. Attribute definition fault

1.1.1.2.1. Faulty initialization value

1.1.1.2.2. Faulty data type

1.1.1.3. Method definition fault

1.1.1.3.1. Faulty return type

1.1.1.3.2. Faulty parameter list

1.1.1.3.2.1. Faulty number of parameters

1.1.1.3.2.2. Faulty data type

1.1.1.3.2.3. Faulty initialization value

1.1.2. Association fault

1.1.2.1. Association at faulty level

1.1.2.2. Multiplicity fault

1.1.2.2.1. Above maximum fault

1.1.2.2.2. Below minimum fault

1.1.2.3. Association property fault

1.2. State fault

1.2.1. Object state fault

1.2.2. System state fault

1.2.2.1. Objects multiplicity fault

1.2.2.2. Object type fault

1.2.2.3. State correlation fault

1.2.3. Class state fault

2. Behavioral fault

2.1. Method Implementation fault

2.1.1. Decision fault

2.1.1.1. Logical condition fault

2.1.1.2. Branching fault

2.1.1.2.1. Early branching fault

2.1.1.2.2. Late branching fault

2.1.1.3. Merging fault

2.1.1.3.1. Early merging fault

2.1.1.3.2. Late merging fault

2.1.1.4. Branches fault

2.1.1.4.1. Missing branch fault

2.1.1.4.2. Extra branch fault

2.1.2. Local statement fault

2.1.2.1. Local assignment fault

2.1.2.2. Local operation fault

2.1.3. Parallelism fault

2.1.3.1. Tines fault

2.1.3.1.1. Missing tine fault

2.1.3.1.2. Extra tine fault

2.1.3.2. Forking fault

2.1.3.2.1. Early forking fault

2.1.3.2.2. Late forking fault

2.1.3.3. Joining fault

2.1.3.3.1. Early joining fault

2.1.3.3.2. Late joining fault

2.1.4. Statement sequence fault

2.1.4.1. Missing statement fault

2.1.4.2. Extra statement fault

2.1.4.3. Statement order fault

2.1.4.3.1. Switched statements fault

2.1.4.3.2. Faulty distance between statements

2.2. Interaction fault

2.2.1. Message fault

2.2.1.1. Message call fault

2.2.1.1.1. Faulty accessibility type

2.2.1.1.2. Argument fault

2.2.1.1.2.1. Switched arguments fault

2.2.1.1.2.2. Faulty value(s)

2.2.1.1.2.3. Faulty number of arguments

2.2.1.2. Return fault

2.2.1.2.1. Faulty return value

2.2.1.2.2. Faulty recipient

2.2.1.3. Wrong object called fault

2.2.1.4. Wrong message sent

2.2.2. Object lifetime fault

2.2.2.1. Construction fault

2.2.2.1.1. Faulty object type

2.2.2.1.2. Late construction fault

2.2.2.1.3. Early construction fault

2.2.2.2. Destruction fault

2.2.2.2.1. Late destruction fault

2.2.2.2.2. Early destruction fault

2.2.3. Sequence fault

2.2.3.1. Missing call fault

2.2.3.2. Extra call fault

2.2.3.3. Call order fault

2.2.3.4. Faulty distance fault

2.2.4. State invariant fault

2.2.4.1. State dependent fault

2.2.4.2. Resultant state fault

Appendix B

Black-Box groups and values for concordance

Group No.	Description	Possible values
1	Help Options	1:-h 2:–help 3:-? 4:None 5:help (without dashes) 6:-help
2	Analysis Options First Character in Switch	1:No switch, or q or n but no p l or s 2:(-p) no l or s 3:(-l) no p or s 4:(-s) no p or l 5:(-?) 6:(-char other than plsqn?) 7:Argument looks like option string but no (-) before the options

Group No.	Description	Possible values
		8:2 of p l and s
		9:All 3 of p l and s
		10:Misplaced option(s)
3	Input File Existence	1:Existing file, no special properties
	And Validity of Name	2:Nonexistent file
		3:No input file name given
		4:Invalid file name
4	Quiet Mode	1:No quiet mode switch
	First or Second	2:Quiet mode switch
	in Switch	3:Switch has second character other than qn
		4:q with separate (-)
		5:q preceded by a space
5	Starting Location	1:No starting location switch
	First, Second	2:Starting location switch, number, no special properties
	or Third inSwitch	3:(-n colon 0)
		4:(-n colon (very large number))
		5:(-n colon (negative number))
		6:(-n colon (non-integer))
		7:Character other than n
		8:Nothing after n
		9:Missing colon between n and the number
		10:No number after the colon
		11:(-n colon real number)
		12:With separate (-) sign
		13:Preceded by a space

Group No.	Description	Possible values
6	Output File	1:Valid output file name 2:No output file name given 3:Invalid output file name 4:abc file already exists 5:wds file already exists 6:Existing file name (with extension other than abc and wds) 7:Both abc and wds already exist or protected
7	Contents of the Input File	1:Empty input file 2:Very large input file 3:Very small word in input file 4:Very large word in input file 5:Unused 6:Unused 7:Special Characters only 8:Normal (nothing special) 9:Contains special characters 10:Unused 11:No input file name given or nonexistent input file 12:Non-English alphabet 13:Unprintable characters 14:All numbers 15:First line is empty
8	Stanza Markers	1:No stanza markers 2:Stanza markers, no integers

Group No.	Description	Possible values
		3:Stanza markers, integers with no special properties 4:Stanza marker integer - negative 5:Stanza marker integer - zero 6:Stanza marker integer - non-integer 7:No input file name given or nonexistent input file 8:Stanza marker integer - real number 9:Stanza marker integer - very large integer number 10:Stanza marker integers - wrong order 11:Syntax correct but put at the end of the line
9	Number of Arguments	1:Unused 2:Too many arguments 3:Unused 4:No arguments 5:Correct number of arguments
10	Control-L in Input File	1:Control-l 2:No control-l 3:No input file name given or nonexistent input file

Table B.1: Black-Box groups and values for concordance

Curriculum Vitae

Name:	Shaimaa Ali
Post-Secondary Education and Degrees:	Alexandria University Alexandria, Egypt 2002 - 2006 M.Sc. University of Western Ontario London, ON 2007 - 2012 Ph.D.
Honours and Awards:	NSERC PGS M 2007-2011
Related Work Experience:	Teaching Assistant The University of Western Ontario 2007 - 2011 Lecturer The University of Western Ontario 2011 - 2012

Publications:

Shaimaa Ali Badawy, Ahmed Elragal and Mahmoud Gabr, ”MSCM: an outlier detection technique for data mining applications”, IASTED Artificial Intelligence and Applications 2008, Innsbruck, 11-13 February.

Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, Wantao Wang, Evaluating the Accu-

racy of Fault Localization Techniques, ASE Automated Software Engineering 2009, Auckland, 16-20 November.

Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani The Application of Data Mining Techniques in Software Fault Localization, CASCON 2008, Toronto, 27-30 October 2008 (Poster Presentation).

Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, Double layers mining for localizing software faults, CASCON 2009, Toronto, 2 5 November 2009 (Poster Presentation).

Shaimaa Ali, James H. Andrews The Faults cube, CASCON 2011, Toronto, 7th - 10th November 2011. (Poster Presentation).