Detecting Dissimilar Classes of Source Code Defects

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Khalid Al Mustansir Billah

©Khalid Al Mustansir Billah, August 2013. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Software maintenance accounts for the most part of the software development cost and efforts, with its major activities focused on the detection, location, analysis and removal of defects present in the software. Although software defects can be originated, and be present, at any phase of the software development life-cycle, implementation (i.e., source code) contains more than three-fourths of the total defects. Due to the diverse nature of the defects, their detection and analysis activities have to be carried out by equally diverse tools, often necessitating the application of multiple tools for reasonable defect coverage that directly increases maintenance overhead. Unified detection tools are known to combine different specialized techniques into a single and massive core, resulting in operational difficulty and maintenance cost increment. The objective of this research was to search for a technique that can detect dissimilar defects using a simplified model and a single methodology, both of which should contribute in creating an easy-to-acquire solution. Following this goal, a 'Supervised Automation Framework' named FlexTax was developed for semi-automatic defect mapping and taxonomy generation, which was then applied on a large-scale real-world defect dataset to generate a comprehensive Defect Taxonomy that was verified using machine learning classifiers and manual verification. This Taxonomy, along with an extensive literature survey, was used for comprehension of the properties of different classes of defects, and for developing Defect Similarity Metrics. The Taxonomy, and the Similarity Metrics were then used to develop a defect detection model and associated techniques, collectively named Symbolic Range Tuple Analysis, or SRTA. SRTA relies on Symbolic Analysis, Path Summarization and Range Propagation to detect dissimilar classes of defects using a simplified set of operations. To verify the effectiveness of the technique, SRTA was evaluated by processing multiple real-world open-source systems, by direct comparison with three state-of-the-art tools, by a controlled experiment, by using an established Benchmark, by comparison with other tools through secondary data, and by a large-scale fault-injection experiment conducted using a Mutation-Injection Framework, which relied on the taxonomy developed earlier for the definition of mutation rules. Experimental results confirmed SRTA's practicality, generality, scalability and accuracy, and proved SRTA's applicability as a new Defect Detection Technique.

Acknowledgements

My deepest gratitude is expressed towards my supervisor, Dr. Chanchal K. Roy, for his guidance, supervision and support that made this research possible.

Besides, I express my gratitude to Dr. Raymond Spiteri and Dr. Julita Vassileva for being in the committee for the evaluation of this research, and to Dr. Ronald Bolton for examining the research as an external examiner.

In addition, a special expression of gratitude goes to Xin Yi, for his implementation of the TXL rules that made the recall measurement experiment possible under the Mutation-Injection framework.

I would like to extend my thanks to all staff of the Department of Computer Science, University of Saskatchewan, for their support for the entire duration of this research in all dimensions possible.

A special mention goes to the members of the Software Research Lab, University of Saskatchewan, for their role as the compassionate and ever helpful colleagues who often came up with casual solutions to problems that seemed infeasible. I would like to mention Muhammad Asaduzzaman and Farouq Al-Omari for their help in revising the draft.

And, as always, I am thankful to my mother for her constant support and encouragement.

I dedicate this thesis to my mother, Naznin.

Contents

List of Tables

List of Figures

List of Abbreviations

FlexTax	FLexible EXtensible TAXonomies
SRTA	Symbolic Range Tuple Analysis / Analyzer
FP	False Positive
FN	False Negative
TP	True Positive
TN	True Negative

Chapter 1

Introduction

With their advancement in time, software systems are becoming progressively complex and this complexity is observed in both their use and structure. Existing systems, through enhancements and improvements, are turning into more complex ones while sophisticated new systems are being developed both as replacements to the older systems, and under completely new requirements. This ever-increasing complexity of existing systems, and the development of new and more complex systems, as apparent from Lehman's assumptions[1] [98], are natural part of the evolution of software technology and are not expected to cease.

As an increasing number of critical processes starts relying on software systems, emphasis on the quality of the software increases to match the trend. To guarantee and maintain a high quality of the finished product, application of quality assurance activities, such as software inspection and testing, is of crucial importance [54] and is enforced through various methodologies.

One of the major tasks in software quality assurance is the prevention, detection and correction of defects present in the system. Defects are defined as artifacts that can occur in any phase of the software development process and are known to cause implications from mere nothing observable to critical failures. Severe complications resulting from software defects include a number of spacecraft accidents [100] and even human casualties in the U.S.A. and Canada [101]. Over the decades, the term *defect* has become an umbrella term encompassing any abnormality introduced or incorporated at any stage of the software development process and is known to be originated by a variety of factors, from simple human errors to complex architectural problems.

Although defect is a valid term to be applied to any phase of software development, implementation (i.e., source code) accounts for more than three-fourths of the total defects found in software systems [99] including at least half of the critical security problems [25, 117]. For the defects that are not originated in the implementation phase, often the most observable traces are found at implementation, a phenomenon explained using the notion, "syntax as a carrier of semantics"[44]. Consequently, detecting defects from source code has become the recipient of much research attention - resulting in the development of a large number of strategies, techniques and tools over the last few decades.

The diversity of the nature of defects has been counterbalanced using equally diverse techniques. The task

[1]A set of eight laws on software evolution published by Lehman and Belady in the period 1974-1996. Three of the laws state that (a) software complexity is always increasing, (b) software size and number will continue to grow and (c) software quality will depend on more rigorous maintenance as the software evolves.

of detecting source code defects has been tackled from many directions, by applying numerous techniques and strategies including various flavours of static, dynamic and hybrid analyses. Each of the strategies, techniques and tools enjoys its own unique combination of advantages, while suffering from an equally unique and diverse set of drawbacks. The narrow focus and specialized treatment of available tools towards particular defects have made their application extremely dependent on specific scenarios. The application of a defect detection endeavour with reasonable defect coverage on a project requires application of multiple tools - often diverse in their nature, input requirements and outputs. Such multiple-tool approaches for defect detection often require much human intervention through source formatting, annotation and input preparations. Detecting multiple defects through a single tool can aid in such scenarios by reducing project overhead and effort requirements. A few commercial tools exist that unify multiple detection activities into a massive internal core (e.g., CodeSonar [63], Coverity SAVE [35], Polyspace C Checker [129], Frama-C [41]) that, in addition to the considerably high cost of acquisition and maintenance, often end up into bizarre errors [29, 99].

Most dissimilar defects show similar characteristics over a certain level of abstraction, as was utilized by most, if not all, of the defect taxonomies and exploited directly in the study by DeMillo and Mathur [44]. Being able to detect such defects using a single abstract model can serve as a low-cost, low-complexity and easy-to-apply technique to simplify the task of defect detection. Such a technique is expected to improve the state-of-the-art in multiple ways including, but not limited to, elimination of the need for different inputs and formats for different defects, elimination of the need for different output interpretations and adaptations, reduction of the complexities of the detection system making it more maintainable and cost-effective, and improvement of the applicability of the process in general.

1.1 Motivation

Software maintenance accounts for a significant portion of the software development activities, often claiming 50%-80% of the total development cost [142, 96] over 80% of the development efforts [139, 2] for successful legacy systems. A major part of the maintenance activities concern the detection and removal of defects that were introduced to the system as "inevitable artifacts" [25]. A failure in the proper detection and removal of the defects can have considerable impact on the quality of the software. A study applying a hypothetical model on real data predicted as much as 33% profit-loss for a moderately sized non-mission-critical software due to improper defect treatment [161].

Automatic defect detection can greatly reduce maintenance overhead and aid in quality assurance, and has been the focus for much research activities. As software is ubiquitous in its nature [93], the defects also tend to get diverse and domain-specific, forcing the detection techniques to be domain-specific and specialized over single or few similar defect types. To provide a reasonable defect coverage, a software needs to be inspected by a number of detection tools coming mainly from proprietary sources. Applications of such diverse tools and techniques also require much external efforts as their input formats and requirements often differ (for example, for the tools that depend on code annotation, a number of different annotation vocabulary and techniques have to be applied).

Unified detection tools (i.e., the tools that detect multiple defects) usually merge the diverse techniques into a single massive core. Such tools tend to be expensive (primary source: [64], secondary source: [86]) and provide with a slow or resource hungry performance portfolio [168]. A workaround to this is to be able to apply a single technique to detect multiple defects. Although a similar approach is used in various flavours over a few existing tools, the application of unified detection so far was restricted over similar defect classes (e.g., different memory defects). Being able to use a technique over dissimilar defect classes thus can greatly simplify the software maintenance activities, providing with attainable and easy-to-use solutions. This research proposes a new technique that combines path summarization, state-space abstraction and symbolic analysis into a scalable, practical and generic defect detection mechanism that can serve as a unified detection approach.

1.2 Research Overview

This section provides an overview of the research questions, describes the problem and solution approach, and describes the contribution of this research.

1.2.1 Research Questions

In this research, the primary objective was to search for a technique to apply to the detection of dissimilar classes of defects. The research questions can be summarized as:

RQ1: Can a specific abstraction provide sufficient means for detecting multiple dissimilar classes of defects?

This question has been treated as the primary research question. A technique was developed with a combination of path summarization, state space-modelling for atomic identifiers, and symbolic analysis that, in both theory and practice, was able to detect a number of dissimilar classes of defects. This research question was answered in Chapter 5 and Chapter 6.

Once the technique was developed, the research then focused on answering the questions that become relevant over the proven existence of such a technique,

RQ2: To what length can such a technique go in terms of dissimilar classes of defects?

After the development of the technique, different evaluation procedures were conducted to assess its effectiveness. The developed technique was found to cover most of the defect classes that do not depend extensively on design or other non-code artifacts. This question was answered in Chapter 5 and Chapter 6.

RQ3: What is the general effectiveness of such a technique in terms of accuracy and performance?

The accuracy of the developed technique was tested through extensive experiments against existing tools and techniques, and against large-scale fault injection procedures. The results confirmed that it is indeed possible to use an unified technique, like the one developed in this research, that would perform as good as, if not better than, the existing tools. This question was answered in Chapter 6.

RQ4: What is the effect of system complexity on the applicability of such a Technique?

A study was conducted to test the effect of the scalability-accuracy trade-off on the developed technique. The technique was found to be resistant to structural and performance complexities of the system it processes, and to scale to large systems without a compromise of performance. This question was answered in Chapter 6 using experimental data.

In order to properly define the objectives, a prerequisite was to establish defect similarity (and dissimilarity) criteria. This was treated using the last research question,

RQ5: What are the criteria and procedure to determine similarity in defect classes?

To answer this question, extensive surveys were conducted on defect classification schemes. Based on the information obtained from the survey, a semi-automated framework was developed to map defects and to create taxonomies in the process. The framework was applied on a large scale real dataset and a taxonomy was generated. The taxonomy was then used to define defect similarity over different coefficients, to define the technique of defect detection, and to define mutation rules used in experiments using the Mutation-Injection framework. This question was answered in Chapter 3.

1.2.2 Problem Description and Solution Strategy

Automated detection of defects from source code is usually done through either of the two code analysis techniques, or their combination. The first of these two, dynamic analysis, analyzes a software's response during execution and is able to uncover defects at a later phase of development [96]. Dynamic analysis is able to provide low false positive rate[2] and is usually more efficient, but usually has high false negative rate[3], requires complete code that can be built and executed, and has no means to analyze execution paths that have not been taken during the particular execution. The second of the techniques, static analysis, analyzes the actual source code. Static analysis usually has low false negative rate, can reach to any execution path through the software but usually suffers from a high false positive rate. As the requirement is to detect defects from source code, static analysis was the compelling candidate for this research.

A problem of static analysis is the phenomenon called "Path Explosion", or the overwhelming number of possible execution paths through a reasonably sized software [60]. Although it is possible for static analysis to check all these paths, it is often infeasible to do a complete check due to the very large set of candidate paths that demands high resources and impractical execution time. To cope with this situation, most static analysis tools reduce the state-space for the software through a compromise between scalability and accuracy, often sacrificing accuracy to scale to large applications.

[2]The ratio of incorrect detection over all detection. Described in Chapter 6.
[3]The ratio of missed detection over all problems present. Described in Chapter 6.

Although static analysis retains the ability to check incomplete code, unless a code is executed, the exact flow of data in the code cannot be determined. To mitigate this scenario a technique named Symbolic Analysis is used. Symbolic analysis is traditionally considered a variant of static analysis as it uses source code to build up symbolic execution values for processing.

Symbolic analysis suffers from the same drawbacks of static analysis, that is, the very large number of paths to check. Techniques often select a subset of the paths to analyze that brings the resource requirements within practical limits. Such techniques often suffer from the cases of a defect hiding in one of the paths that were excluded. On the other hand, as symbolic analysis uses inferred or abstract values instead of the real data, it provides the flexibility of applying arbitrary abstraction to the analysis process. It has been proven that if symbolic analysis can be used as a compositional tool, it can alleviate the path explosion problem effectively [59]. One of the objectives of this research was to utilize this property of symbolic analysis to overcome its own limitations that has been achieved using path summarization - a novel application of compositional analysis.

Defects in source code often manifest themselves in a local context [96]. This nature of defects requires path and context sensitive analysis techniques to be applied to make the detection effective and efficient. The requirement of path sensitivity, and the nature of defects in staying in local contexts, are outlined in a recent research work by Le and Soffa [96]. Again, to address this issue, symbolic analysis plays the crucial role due to its flexibility to abstraction. To provide path sensitivity, context-awareness and wide-expanse capabilities, symbolic analysis was chosen as the analysis technique.

Often balancing the scalability-accuracy trade-off results in sacrificing one in lieu of the other. Many of the widely used tools sacrifice accuracy to achieve scalability [163]. Still, only a few tools ensure good scalability [168]. In this research, the target was set to provide better accuracy, without deliberately sacrificing scalability. The solution proposed by this research is realized through a framework, SRTA[4].

Figure 1.1 describes the solution in a three-layer diagram, the top being the research goals, middle layer showing the specific components of the solution towards achieving that goal and the bottom layer listing the results obtained in the particular objectives.

Main goals for this research can be summarized into four parts - Accuracy, Scalability, Generality and Practicality. The term 'Accuracy' is used to denote both a low false positive and low false negative rate. Path summarization, context-sensitivity, and complete path envelopment in a form of inter-procedural analysis have been used to tackle the problem of accuracy. Path summarization acts as a countermeasure to false negative errors as it does not exclude any path from the analysis, as well as balancing the path explosion problem by introducing summarized representations. Context-sensitivity and path envelopment both act in favour of reducing false positives by taking the context into account, and by considering the exact sequence of events that led to an entity's states in the program.

'Scalability' of a system denotes the ability to retain its qualities against a varying range of properties.

[4]Abbreviation for Symbolic Range Tuple Analysis / Analyzer. Details are provided in Chapter 5.

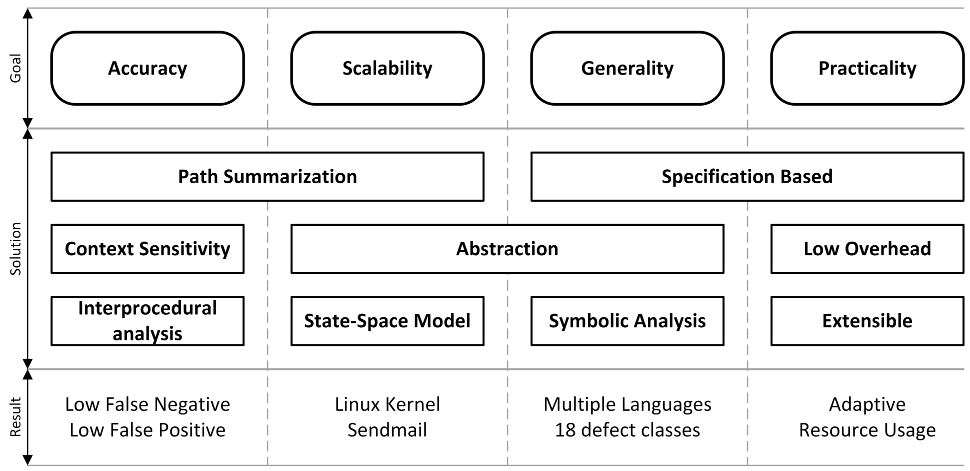


Figure 1.1: Solutions Provided by the Approach (Adapted from the Dissertation of Wei Le, 2010)

In this context, scalability is used to denote the capability of successful processing with codebases of various sizes, and with various degrees of information that can be inferred from the codebases. A scalable technique should be able to process small and large software systems alike. Path summarization, as used in this technique, contributes in ensuring scalability by simplifying and reducing the path explosion problem. The state-space model used with path summarization and symbolic abstraction also contributed in making the system scalable.

'Generality' is used to describe the technique's ability to be adapted in different situations. The mechanism proposed in this research does not use source code directly, rather converts the code into a simplified intermediate model related to the software state-space, and uses symbolic analysis techniques instead of pure static analysis. This approach ensures the adoption of the technique over different software systems, different languages and across different degrees of completion for the systems.

'Practicality' is the applicability of the technique over real-world scenarios, that is, the ability to function within reasonable environmental bounds. The technique developed in this research is practical in the sense of resource requirement and execution time, covering more defect classes as compared to most existing tools and in an execution environment with reasonable bounds. The state-space abstraction and completely soft-coded specifications aided in making the system practical.

1.2.3 Contribution of this Research

(1) The primary contribution of this research was to develop a symbolic analysis technique, SRTA, to detect dissimilar classes of software defects. The technique relies on path summarization and state-space abstraction and was proven to be applicable on dissimilar defect types, incorporating generality and scalability.

(2) After the development of the technique, extensive experiments were conducted to verify its aspects. The experiments were performed on widely used real-world open-source projects, against a benchmark suite, through a large scale fault-injection procedure involving a Mutation-Injection framework, and in a controlled

experiment devised to check resource consumption. The experiments confirmed the practicality of the technique.

As a prerequisite to the research, a measure was required to establish defect similarity, which, as its requirement, relied on a custom developed defect taxonomy. The other three contribution of this research was in achieving this goal.

(3) A framework, FlexTax, was proposed to develop extensible defect taxonomies. The framework is a new concept in that it deals with *extensible* and *flexible* taxonomies and utilizes objective similarity metrics to classify defects. The concept addresses at least three of the major issues related with taxonomy generation - orthogonality, completeness and non-redundancy.

(4) A defect taxonomy was developed using FlexTax and later used in comparison and assessment of the different tools and techniques described in this dissertation. The taxonomy was validated against real world data and was found to be flexible, extensible, complete, orthogonal and non-redundant.

(5) Based on the Taxonomy, defect similarity was evaluated against custom designed, and industry standard similarity coefficients. The evaluation was able to point out similarity criteria that were used in the research.

1.3 Organization of this Dissertation

This dissertation is organized in seven chapters, with the current chapter being the brief introduction to the rest.

Chapter 2 provides the background to this research. It includes definition of common terms used in the dissertation and provides a brief description of the research methodology.

Chapter 3 Provides the description of the prerequisites to the research - that is, a taxonomy generation framework, a taxonomy, and defect similarity measures. The chapter describes the most relevant among the existing categorizations of defects and analyzes their properties. The chapter then proceeds on proposing a framework for developing extensible defect classifications and describes a defect classification scheme developed using the framework. Tests and analyses were conducted using the taxonomy before adopting it as the taxonomy for the rest of this research, and the data from the tests and analyses is presented in this chapter. Finally, the chapter describes the measures for defect similarity and establishes common similarity criteria.

Chapter 4, using the taxonomy established in Chapter 3, discusses the current state-of-the-art. It describes and compares relevant tools and techniques that have been employed by defect detection activities in the academia and industry. The techniques and tools were analyzed with respect to their reported strengths and weaknesses. Chapter 4, along with the first part of Chapter 3, contributes to the literature review for this research.

Chapter 5 gives the details of the proposed mechanism, SRTA, with its symbolic framework. The chapter details the specifics of the adopted symbolic analysis, abstraction and the underlying model. The chapter then proceeds on describing the use of SRTA in detecting dissimilar classes of defects. The chapter includes brief description of a prototype implementation of SRTA.

Chapter 6 details the experimental setup, data, and analyses on the data to assess the developed defect detection technique. The technique is compared with existing tools using open-source projects, and through a large-scale fault-injection experiment conducted with the aid of a Mutation-Injection framework.

Chapter 7 concludes the dissertation.

Chapter 2
Background

This chapter provides the background to this research. In Section 2.1, common terminology used in this dissertation is provided. The next section, Section 2.2, provides an overview of research methodology and research phases, including notes on information source selection, data acquisition and analysis, and specific development information.

2.1 Terminology

This section describes the general terminology used in this dissertation. In addition to the ones presented in this section, relevant terms and definitions are included in appropriate places in later chapters of the dissertation.

2.1.1 Defect

Software defect has been the subject of multiple definitions. The definitions come from individual research, as well as from standardization organizations like IEEE.

Fault

A fault is an accidental condition that causes a functional unit to fail to perform its required operation [73, 122]. Often the fault leads to a failure [122] thus acting as the underlying cause of the failure.

Failure

A failure is an event where a system or a component of the system stops performing a required function within acceptable boundaries [73, 122]. Failures are often classified as the breakdowns of the system functions or the observable points of deviation from its expected course during execution. In summary, it is the observed impact of faults.

Error

An error is the human action that results into a fault in the software - it can be an omission or misinterpretation of user requirements in the software specifications, incorrect translation, or omission of a requirement in the design document [73, 122]. In summary, an error is the action that creates a fault.

Adopted Definition of Defect

Often the term "defect", analogous to "fault" and "bug", is defined differently by different sources - including the mention as "a product anomaly" [73] or "the imperfections found in the code during technical reviews" [122]. In the context of this study, a more elaborate definition relevant to source code is adopted.

Our adoption of the term *source code defect* indicates defects detectable from source code. In extended definition, the term is used to indicate an artifact that,

(a) Exhibits its presence in source code, regardless of the origin,

(b) Is not a result of a language or environmental constraint beyond the control of the developer.

(c) Either produces wrong output or deviates the behaviour of the software from its expected course,

(d) Or provides a way to externally initiate or exploit the (c) above.

Throughout this dissertation, the term 'defect', unless mentioned otherwise, was used to indicate the source code defect as described in this section. Clause (a) of this particular definition for defect was adopted to ensure the defect come from source code only (that is, not policy problems or architectural violations). Clause (b) excludes the cases where the developer was restricted by unsolvable constraints by the language or environment. Clause (c) defines the generic nature of the defect, for proper identification. Finally, clause (d) was included to include vulnerabilities in the set of defects.

Defect Location

Defect location is the process of isolating the specific locations or regions of source code that contain the defect footprint and have to be modified for the removal of the defect [88]. Defect location complements defect detection by adding the location information with the presence of defects. The location can be reported in various granularities, to statements, functions, class definitions, files or modules.

2.1.2 System Qualities

Scalability

The term "Scalability" is defined for a system as, "a quality of software systems characterized by the causal impact that scaling aspects of the system environment and design have on certain measured system qualities as these aspects are varied over expected operational ranges" [46]. A scalable system is a system that can accommodate this variation in a way that is acceptable to the stakeholders [46].

In the present context, the phrase "aspects of the system environment" means the subject system's size that can be expressed as any software size metric (e.g., LOC[1], Complexity, NASA's Object Oriented Quality Metrics [123]). The "measured system qualities" denote the capability of successful detection, with varying size of the codebase and inferred information.

For this research, lines of code was adopted as the size metric and scalability was considered over a range of the software LOCs.

Accuracy

Accuracy of detection refers to detection of the relevant artifacts. In determining the accuracy of detection, two established measures were used in this research.

One of the measures, Precision, denotes the probability that a detected artifact is relevant [61]. That is, out of all detected artifacts, it specifies the probability of any artifact's being a relevant one. In this context, the term "relevance" indicates that the detected artifact contains the properties it is expected to contain (i.e., a defect has the properties that make it a defect). The other measure, Recall, expresses the probability that a relevant artifact is detected among all present artifacts [61]. The two associated terms, false positive rate and false negative rate, are alternate ways of expressing precision and recall. A false positive rate is the portion of artifacts that were detected incorrectly, over all detected artifacts (thus, 1 - false positive rate is the normalized precision) and a false negative rate is the portion of artifacts should be detected but are not, over all existing artifacts (thus, 1 - false negative rate is the normalized recall).

Detailed information on precision and recall, including the mathematical expressions used to estimate their values and the specific procedures followed in this research, is presented in Chapter 6.

2.1.3 SRTA Specifics

This section describes the terminology specific to this research.

Artifact

The term artifact is used in this dissertation to mean any significant entity in the source code. The definition spans to variables, constants as well as to statements and language constructs.

Entity

An entity is an artifact in the software that participates in operations. An entity can be an object (for object-oriented / object-based systems), a variable, a constant, an array, a reference or a pointer.

[1]Lines of Code. Usually counted excluding comments. Two variants are in widespread use. (a) S-LOC, or Source Lines of Code is the number of linebreaks in the source, and (b) L-LOC, or Logical Line of Code, is the number of logical statements in the source

Atomic Entity

An atomic entity is an entity that cannot be decomposed into a collection of same or different entities. Any entity with the basic type permitted in the programming language is an atomic entity.

Collective Entity

A collective entity is an entity that acts as a common reference point for a collection of atomic entities. Any array or a pointer (for C / C++) that points to an array is a collective entity.

Composite Entity

A composite entity is a collection of atomic entities grouped under a common name. Any structure / class / union is an example of a composite entity.

2.2 Research Methodology

This section describes the methodology used for the research. Different phases of research involved the following,

(i) Literature survey.

(ii) Problem selection, specification and analysis.

(iii) Framework development for defect classification.

(iv) Experiment concerning the framework for defect classification.

(v) Model generation and analysis technique development.

(vi) Experiment and analysis under the developed model and analysis techniques.

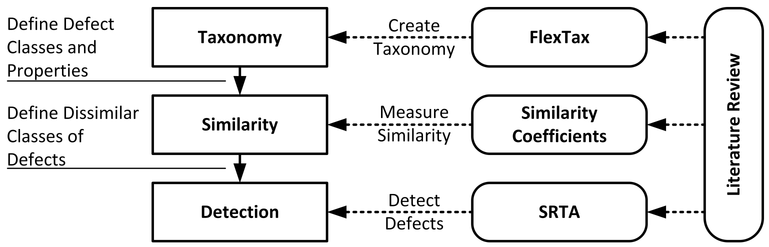


Figure 2.1: Solution Phases for the Research

Development of the solution was performed in three distinct stages, as showin in Figure 2.1. In the figure, rectangles denote particular solutions and the rounded rectangles denote the approach or technique for the solution. The main goal of this research is to search for a technique for detecting dissimilar classes of defects.

The realization of this problem requires a definition on 'dissimilar classes of defects', which was provided by the defect similarity study. The defect similarity study, for its functioning, requires a suitable defect taxonomy for the definition of defect classes and their properties. The bottom-up processing order of the problems in the figure denotes the sequence of the steps.

Solutions to these problems are presented in the top-down approach on the same figure. FlexTax was developed as a semi-automatic framework to create defect taxonomies. Once the taxonomies were generated, similarity coefficients were applied to measure the similarity (and dissimilarity). These similarity measures were then used to develop SRTA to solve the problem of detection. Literature Survey was involved in every phase of the research.

2.2.1 Literature Survey

As the first part of our research, the approach to a systematic literature review [19, 80, 118] was followed to collect, analyze and process information. We have conducted the step-by-step formulation of the literature study to make it unbiased, complete and comprehensive. The procedures we have followed in this step comprise to the mandatory parts of the Planning-Conducting-Reporting methodology recommended by Kitchenham et al. in their 2009 paper [80]. The steps included,

(i) Planning the review

  (a) Identifying the need for a review

  (b) Specifying the research questions

  (c) Developing a review protocol

(ii) Conducting the Review

  (a) Identification of research

  (b) Selection of primary studies

  (c) Study quality assessment

  (d) Data extraction

  (e) Data synthesis

(iii) Reporting the outcome

  (a) Specifying dissemination mechanism

  (b) Formatting the main report

Table 2.1: Inclusion and Exclusion Criteria

Property	Item	Inclusion Criteria	Exclusion Criteria
Type	Research Articles	Peer Reviewed Technical Reports	Position and Student Papers
	Magazine and Newspaper Articles	Objective Reports	Subjective Opinions
Content	Tools / Techniques	Address Source Code Defects Applicable to Source Code Defects Adaptable to Source Code Defects	Do not apply to Source Code Defects Deprecated Proven Invalid
Relevance	Year of Publication	On or after 1990	Before 1980 (with notable exceptions)

Information Source Selection

We have used well known publication repositories to collect the published information that corresponds to the research questions. In the fields of relevance, the ACM Portal and IEEE XPlore have been sources of interest for their large collection of published articles and ranking flexibilities. We have complemented the search using Google search and Google Scholar search to find relevant publications. Our effort built a repository of more than 200 published artifacts that we have analyzed to prepare this study.

Our searching methodology was twofold. First, we searched the repositories for predefined search strings that correspond to our research questions. Our search strings were formulated using original form, alternate phrasings and synonyms for the structure (“bug”, “fault”, “defect”) (“”, “detection”, “analysis”). Second, while exploring the papers that were collected from the first methodology, we have marked and collected the relevant papers that did not turn up in the first search but were referred in the papers we have been exploring.

Primary Studies Selection

To collect the data and information, we selected the publications for consideration based on defined inclusion and exclusion criteria as summarized in Table 2.1. Our studies resulted on a large publication repository that we have rigorously analyzed, categorized and included in this study. The studies conducted using the materials are presented in Chapter 3 and Chapter 4.

Data Extraction and Analysis

We have collected the data from the publications that primarily reported the data on a tool or technique, or performed an experimental study involving the tool or technique and reported data from the experiment. The biggest ordeal for us was that, not all publications report the data on the parameters we are interested in, and although sometimes it is possible to map or derive the data from the ones provided, there also are situations where no direct data was provided by the publications to use in assessing specific tools and techniques.

2.2.2 Problem Selection and Analysis

After the comprehensive analysis of the literature, we identified the issues and open problems relating to defect detection, which defined the need for a unified detection approach for software defects and set the objectives for this research. From the information we analyzed, it was clear that defect similarity metrics need to be established before approaching the main problem of unified detection. Consequently, the prerequisites for the main objective emerged as the establishment of a defect taxonomy and similarity metrics under a defined framework.

2.2.3 Taxonomy Generation Framework and Taxonomy

To focus the research effort, a taxonomy on source code defects was necessary. Existing taxonomies, although vast in numbers, do not usually provide separate treatment to source code defects. Thus, we required a taxonomy to be constructed to cater for our need, additional to the requirement of defect similarity assessment.

Instead of generating a specialized taxonomy suitable only for the current research, we have instead proposed a framework for generating extensible taxonomies to suit general needs. The framework is generalized, flexible and automated to generate complete, orthogonal, non-redundant and abstract taxonomies from earlier taxonomies or real defect data or a combination of both. The framework was used to generate a taxonomy out of a large quantity of real defect data and evaluated against real data sources following multiple procedures, including the use of machine learning classifiers and manual analysis. The taxonomy generated by the framework was then used to analyze and compare existing tools and techniques with the one developed. The Framework was described in Chapter 3.

2.2.4 Model Generation and Analysis Technique Development

To facilitate detection of dissimilar defects, source code needs to be incorporated into an abstract model that enumerates necessary features required for the detection. A symbolic analysis model using range tuples was designed for the purpose.

The model was used in multiple customizable analyses to detect the presence of the defects. The model and the analyses were developed with the objective of generating high precision results with considerable scalability across software sizes and implementation technology. The model and analysis techniques were described in Chapter 5.

2.2.5 Experiment and Analysis

The model has been realized with a working prototype and applied on real-world systems to assess its generality, detection capability, accuracy and scalability. The test systems were chosen from widely used open-source projects and from diverse functional domains.

To incorporate a comprehensive testing of the system, the experiment was performed in multiple different sub-experiments. First, the prototype was applied on open-source software systems implemented across two general purpose languages and the accuracy of detection was analyzed. To assess the false negative performance, the prototype was evaluated against a benchmark, and was separately evaluated using a large-scale fault injection experiment. Additionally, the prototype was compared against other state-of-the-art tools through direct experiments, and through secondary data.

For each of the sub-experiments, a case-by-case analysis was performed to describe the prototype's behaviour, limitations and advantages. Chapter 6 describes the experiment and analysis.

2.3 Summary

In this chapter, common terminology, and an overview on research methodology were provided. The chapter started by providing the definition of common terminologies as they are used in this research and then continued on describing the research methodologies and phases. The methodologies provide details of information sources, inclusion and exclusion criteria, research phases and sequence and analysis techniques.

Chapter 3

A Taxonomy of Defects and Defect Similarity

This chapter describes a taxonomy of defects and a study on defect similarity. The chapter begins by a section on the chapter-specific background (Section 3.1), and then continues on with a description of taxonomy structures (Section 3.2) and the trend of taxonomy development with their desirable properties (Section 3.3, Section 3.4), following the methodology outlined in Chapter 2. Using the notion of the outlined properties, descriptions of notable defect taxonomies are provided next (Section 3.5). Next, aiming on utilizing the analyzed features of defect taxonomies, a framework named FlexTax is proposed and described along with the details of its underlying model and analysis techniques (Section 3.6). Next, the description proceeds in detailing the experience of using FlexTax over the 25k real-world defect data of the CVE [119] dataset (Section 3.7). This case study generated a taxonomy of software defects which is described in details using examples and formal definitions (Section 3.7.3). This taxonomy's evaluation using machine learning classifiers and manual analysis against established databases is described in the next section (Section 3.8). Finally, using this taxonomy, and considering the underlying principle of FlexTax, a study is described to find defect similarity (Section 3.9).

3.1 Chapter-specific Background

This section provides chapter-specific terminology, definitions and descriptions, in addition to those provided in Chapter 2.

Defect Class: A defect class, as it is used throughout this dissertation, is the description of a specific phenomenon that has resulted from the underlying causes that count as defects. The defect class is the description of the phenomenon, rather than the exact anomaly that results into the defect. One defect class may contain one or more defect types. An example for a defect class is the 'Undefined Outcome', that describes a situation where the outcome of an operation cannot be determined.

Defect Type: A defect type is an exact anomaly that is considered as a defect. A defect type describes one specific problem in the code that, in effect, creates the detectable defect. One defect type contains more than one defect instances. Continuing the example defect class from the previous definition, a defect type under the class "Undefined Outcome" can be a division-by-zero, the use of an uninitialized value in an operation, or performing undefined arithmetic operation.

Defect Instance: A defect instance is the occurrence of an anomaly, belonging to a defect type, in the code. Every single occurrence of an anomaly counts as a separate defect instance. Continuing the example from the previous two definitions, every single division-by-zero instance counts as a defect instance.

As an example to illustrate the three terms, if a software's code contains ten divisions-by-zero and ten uninitialized value usage, it should count as twenty defect instances, two defect types and one defect class.

3.2 Taxonomy and Taxonomy Structures

A taxonomy is a classification system that allows one to uniquely identify the subjects of interest, often depicting the subject of interest and its classification category as an ordered tuple [13]. In other words, a taxonomy assigns a set of defects to a set of distinct classes or categories under a given rule. Taxonomies can be formed for multiple purposes, and in three different structures, as reported by the literature.

3.2.1 Flat or Non-hierarchical Taxonomies

Flat taxonomies ideally contain one single set of distinct categories without any overlapping criteria. Such taxonomies realize only one point-of-view at a time.

The advantages of flat taxonomies are the easier generation and simplified decision making - as there is only one level of hierarchy, a new class can be added with the consideration of basic properties. The strongest disadvantage of such taxonomies is the restriction on the freedom of interpretation. As such taxonomies (e.g., [82, 44, 10]) are developed only through one point-of-view, most often that of the developer, they are usually not adaptable to other's needs.

3.2.2 Hierarchical Taxonomies

Hierarchical taxonomies arrange the categories in different levels, where each level generalizes the levels below it and specializes the levels above it. Hierarchical taxonomies can (but do not always) realize more than one points-of-view at a time.

The most prominent advantage of hierarchical taxonomies is the increased degree of freedom of interpretation. The disadvantage of the hierarchical taxonomies is the more complex structure than the flat taxonomies. Addition of a new class has to be decided from a number of different directions, should it contain more than one views. Additionally, such taxonomies (e.g., [6, 116, 162, 99, 140, 120]) tend to suffer from difficulty in completeness.

3.2.3 Matrices

Matrices arrange different perspectives or properties through different directions of arrangement, and use the combinations of the cross-cutting directions to define defect categories. Matrices always realize more than one points-of-view for classification.

The advantage of matrices is that, they fully realize different points-of-view, providing room for later interpretation. The disadvantage is that, based on dimensions, matrices can be complex to map into. The addition of a new class on a matrix is dependent on the interaction of a number of different points-of-view. Such taxonomies (e.g., [62, 28, 89]) tend to be the most flexible and extensible.

3.3 The Trend of Defect Taxonomies

In commercial software development projects, statistical fault density metrics act as tools of cost estimation and decision support [128] and often suffice in achieving the goals. However, it is argued that a comprehensive defect taxonomy is a necessity, and not just a supportive measure, for proper software quality assurance activities [128, 140].

A number of defect taxonomies are in existence and being used, varying both in their content and intent - focusing in areas such as errors, behaviours, functionalities, vulnerabilities, and, incidents and attacks [156, 155]. The diversity of defects and their specialized attachments to different projects made it difficult to apply the same set of defect classes to all, or most, projects and thus has contributed to the large set of parallel taxonomies in existence today. None of the classifications has become a truly and broadly applied practice, with their practical application restricted mainly on severity or impact only [159]. The taxonomies are used much discretely by different organizations, sometimes having different branches of the same organization adopting different taxonomies to describe the same defects [140]. The reasons for the emergence of parallel taxonomies are attributed to different factors by the literature, including the diversity of data sources and formats [140], independent applications of quality assurance techniques [54], lack of orthogonality [54, 140, 124], use of parallel terminology [140], ambiguities or unclear descriptions [54], non-standard documentations [54] and a learning curve associated with defect categorization at the early stages of the projects that compel developers to record defects without properly classifying them [140]. Nevertheless, whatever the reasons for not having one may be, the need for consistent defect taxonomies can neither be denied nor ignored.

Although a few taxonomies like the Hewlett-Packard Defect Taxonomy and the Orthogonal Defect Classification Scheme have established themselves as prominent tools for defect analysis [54], it is difficult, if not impossible, to develop a single taxonomy for all software systems of the world due to their extreme diversity in technologies, objectives and tasks. Existing defect taxonomies for the implementation artifacts (i.e., source code) have the same limitations, as it is apparent from multiple taxonomies that are in effect today. Although the complete taxonomies that span all phases and artifacts of a software life-cycle differ due to the extremely diverse tasks a software system achieves, different source code defect taxonomies do share similar elements that can be generalized into a higher level of defect classifications with widespread applications.

3.4 Criteria for Developing a Taxonomy

A number of factors have driven the defect classification objectives. Schemes have been developed focusing on defect origin [89], essential functionalities [82], effects [89], activities to fix the defect [44] and the impact the defect may cause [6]. Although different criteria are considered for different defect taxonomies and there is no unanimous agreement by researchers about the complete set of properties that will identify a perfect taxonomy, a number of desirable qualities were established and agreed upon by most of the concerned.

3.4.1 Desirable Properties

A study by Freimut [53] investigated the properties of defect taxonomies, along with those of the defects, and made a number of recommendations that afterwards have made their way into notable research in the field [140]. According to Freimut's original recommendations [55] and a later follow up study [53], a defect taxonomy should contain at least five desirable properties - Clearly Defined, Exemplified, Orthogonal, Complete and Small in Number of Categories.

Clearly Defined: Each category in the taxonomy must be clearly and precisely defined [53]. This quality, if ensured, can eliminate a number of problems prevalent in existing taxonomies, including the ambiguity and strictly subjective usage.

Exemplified: Each category should be provided with example(s) [55, 53]. This recommendation aims to establish that no hypothetical category is included, ensuring the non-redundancy of the taxonomy.

Complete - The categories should be complete in a sense that, a category exists for every defect encountered along the process [53].

Orthogonal - Categories should be mutually exclusive, so that only one category is applicable to a defect [53].

Small Number of Categories - The number of categories should be small, typically from five to nine [54]. We disagree with this particular recommendation as it is not always possible to ensure this property over different points-of-view, and on different projects. Furthermore the term "Small Number", due to its subjective definition, can lead to confusion in taxonmies.

Additional to this set of properties, other researchers [12, 72, 107, 87] have proposed a number of properties for good taxonomies that comply with this set of properties. A number of researchers seem to converge in one opinion - that taxonomies should describe the properties of the elements they classify, and not the individual problems themselves [111, 87, 12, 13].

3.4.2 Additional Properties

In addition to the proposed properties of taxonomies, we argue that two more properties need to be complied with to make a taxonomy effective and useful.

3.4.3 Flexibility

By the term 'Flexibility', we mean the ability of a taxonomy to accommodate itself with changing situations without altering its base structure. In other words, flexibility means the ability of the taxonomy to reconfigure its defect mapping based on the need. The rationale behind this argument consists of two points. First, while developing a taxonomy, a developer may find situations that require a reinterpretation of previous decisions. If a taxonomy is not reconfigurable, this reinterpretation usually requires discarding the unfinished taxonomy and develop another from scratch. A reconfigurable taxonomy can avoid such workload by accommodating the new interpretations and by remapping the defects as required.

The second argument is, as taxonomies are developed to serve subjective interests [90], often the taxonomy developed by one do not suffice for the need of another on a similar endeavour. If the taxonomy is flexible, the later user can reconfigure it according to the need and make it a better suited device for the task.

3.4.4 Extensibility

'Extensibility' is a common term in Software Engineering used to indicate the property of a software system of being extended to encompass new requirements. In the present context, it is used to denote the means to add new categories to taxonomies should the need arises. This property is required because, due to the ubiquity of the nature of the software, it is not often possible to predict what situation may arise in future, for other software or for the future versions of the same software. Such unforeseen situations also bring in yet un-encountered defects. If a taxonomy does not provide a mechanism to extend itself to the need of the new requirement, it runs the risk of being obsolete. In such cases,the efforts involved in developing a taxonomy may be required to repeat for developing a new taxonomy for the same purpose for every new release of the same software system. To preserve the consistency of the taxonomies, extensibility must be ensured over defined and formal rules.

3.5 Existing Taxonomies

From an overwhelming number of present taxonomies, twelve taxonomies were chosen for analysis of their properties. The taxonomies were chosen if they provided a specific treatment to source code defects, and if they complied to any of the four additional criteria - (a) Developed in recent time (within 5 years), (b) Received considerable citations, (c) Reported to have considerable success either in the industry or in academia and (d) Provided an unusual treatment of the problem.

The discussion on the taxonomies include the reason behind its selection, a general description and the taxonomy's strengths and weaknesses.

3.5.1 Taxonomies Focused on Specific Software

This section describes the taxonomies that apply to specific software only. The taxonomies discussed in this section were either developed using a single software system, or were developed to be applied on a single software system.

Errors of TEX

Published by Knuth [82] in 1989, a description of errors in the popular typesetting system TEX created one of the earlier examples of defect taxonomies. This classification scheme was based on essential functionality rather than being based on the external observable artifacts of the program [6, 82].

The Errors of TEX creates categories with the names, "An algorithm gone awry", "A blunder or a mental typo", "A clean up for consistency or clarity", "A data structure debacle", "An efficiency enhancement", "A forgotten function", "A generalization or growth of ability", "An interactive improvement", "A language liability", "A mismatch between modules", "A promotion of portability", "A quest for quality", "Reinforcement of robustness", "A surprising scenario" and "A Trivial Typo" [82, 6].

However, this classification is focused on TEX system only and is difficult to apply on systems where the original code writer is not the one applying the classification [87], making it strongly subjective. The taxonomy is complete till the time it was applied, with no possible extension in future - thus having a tendency towards incompleteness once a new feature is added or the software is reimplemented. It is strongly non-orthogonal as multiple categories overlap with each other.

DeMillo and Mathur's Classification

DeMillo and Mathur [44] proposed a grammar based defect classification scheme as the basis of an automatic defect classifier and applied it on the Errors of TEX [82]. Their scheme classifies a defect into one of the four major classes - "Missing entity", "Spurious entity", "Misplaced entity" and "Incorrect entity" where an entity is a representation of the defect [44]. Among their classes, missing entities are the ones that require the introduction of an element in the code to correct it [44], encompassing all kinds of omitted checks and cleanup codes. The second of the four, spurious entity, indicates those faults that require a removal from its characteristic elements from the code [44] - thus many security flaws, code clones [137, 133] and performance bugs qualify for this category. Third in the category, misplaced entity, denotes those cases where an element is required to be relocated inside code to remove the defect [44] and the fourth and last category, incorrect entity, is described simply as the defect that cannot be classified into any of the other three [44].

This taxonomy is strongly objective, requiring minimum human supervision, and no subjective discretion, in its generation process. However, the taxonomy suffers from an extremely concise structure involving four categories only, which may suffice for a single system like TEX, but are not applicable to larger and more diverse systems like the Unix.

3.5.2 Taxonomies Focused on Organizations and Processes

This section describes the taxonomies that were aimed at specific organizations or engineering processes.

The Hewlett-Packard Scheme

The Hewlett-Packard Taxonomy includes all phases of development [54]. A defect is described using three attributes - “Origin”, “Type” and “Mode”. Origin refers to the phase of introduction for the defect, type describes a particular origin in more details and mode describes the reason for which the defect is considered a defect [54]. The choice of origin specifies a set of values that further elaborates the origin [53].

In the predefined six choices for origin, only one is dedicated to code. The types associated with code draw from the set containing “Interprocess Communications”, “Data Definition”, “Module Design”, “Logical Description”, “Error Checking”, “Standards”, “Logic”, “Computation”, “Data Handling” and “Interface Implementation” with the first six shared with design defects [53]. For each of the types, a single attribute from the list of “Missing”, “Unclear”, “Wrong”, “Changed” and “Better way” can be assigned [53].

The HP Taxonomy provides a generic structure applicable across software development companies in their tasks, and is generic to be adopted to other granularities. But the taxonomy is neither flexible nor extensible, limiting its applicability beyond the range of application software.

The Orthogonal Defect Classification Scheme

Introduced by IBM, the Orthogonal Defect Classification scheme (ODC) was developed focusing on the defects found in code [54]. ODC associates a defect with a set of orthogonal attributes that identify the process requiring attention, in the same way of characterizing a point using n values in an orthogonal n-axis Cartesian space [28].

The ODC identifies eight defect types to be applied on the processes [28]. Coding is described as a process thus the defect types do apply to code defects, along with their relevance to other artifacts, processes and phases of the software life cycle. The scheme focuses on the existential attributes of defects instead of the inference based approximate location of its injection [28].

Of the eight defect types, the first is “Function”, that denotes a problem in a new feature added to the software. The second type is “Interface”, that accounts for both the internal and external interfaces of the components. The third defect type, named “Checking”, refers to the validation rules in the code. The fourth type, “Assignment”, denotes all kinds of data assignment operations. The fifth type, “Timing / Serialization”, denotes the defects that occur in concurrent or distributed programming where resource sharing and parallelism are in effect. The sixth type, “Build/Package/Merge”, includes defects in build systems, libraries, version control and migration. The seventh type, “Documentation”, is the set of defects that affect both publications and maintenance notes and the last of the types, “Algorithm”, describes a code defect in its pure form - any logical or arithmetic error that affects the efficiency or effectiveness of the software and can be fixed by correcting or at best, re-implementing a module [28].

A notable point about the ODC is that it is not a concrete or fixed defect classification scheme. Rather it is more of a concept that was instantiated through the eight defect types, and that includes room for any adoption in compliance with its basic requirements, the requirements being orthogonality, consistency across phases and uniformity across processes [28].

3.5.3 Taxonomies Focused on System Aspects

This section describes taxonomies that were developed on generalized system properties, instead of a specific software, engineering process or organization.

Unix Security Taxonomy

Published by Aslam in 1995 [6], this taxonomy focuses on security faults present in Unix Systems and takes an approach to classify defects through software fault analysis. The faults are classified into three distinct groups - "Operation faults", "Environment faults" and "Coding faults", of which coding faults are directly relevant to our current study. This taxonomy relies on a defect database [6] and employs objective decision making processes [13].

One strict criticism of this taxonomy is that, despite the incorporation of real world data, the taxonomy was designed for implementation specific UNIX faults only [13]. A second issue, as pointed out by Bishop [13], is that the Unix Security Taxonomy contains very prominent ambiguities due to its insufficient specification of the point-of-view.

Extracted from the taxonomy, we found the relevant-to-code elements to be two types of coding faults, the "Condition validation error" and "Synchronization error". The condition validation error is defined to contain two categories, "Failure to handle exceptions" and "Input validation error", with the latter further subdivided into "Field value correlation error", "Syntax error", "Type and number of input fields", "Missing input" and "Extraneous input" [6, 156]. Synchronization errors contain "Improper or inadequate serialization" and "Race condition".

Landwehr's Program Security Flaw Taxonomy

Landwehr [89] developed a taxonomy from three different angles - "Genesis", "Point of introduction" and "Location" [156]. Of the three, a number of defects correspond to the source code defects. These defects, even if they do not get exploited as security flaws, can cause improper behaviour by the software and thus fall into our interest. Categorized under the super-category "Inadvertent Flaws", these types include "Incomplete or inconsistent validation", "Domain error", "Serialization or aliasing", "Inadequate authentication", "Boundary condition violation" and "Exploitable logic errors" [89].

This taxonomy is specific in its task, focusing on security flaws only. Although its three-tier consideration of the defects provides a different approach towards the classification problem, it is not applicable beyond small-scale desktop systems.

Basili and Pericone's Taxonomy

Published in 1984, Basili and Pericone's Software Fault Taxonomy classifies the faults based on environmental factors [10]. Although the taxonomy suffers from a few ambiguous definitions [44], it provides five classes that all relate to code defects. The classes of defects are mentioned as "Initialization", "Control structure", "Interface", "Data" and "Computation" [10, 44]. This taxonomy is notable for its unusual treatment of software defects in relation to environmental cause.

3.5.4 Classification of Unusual Code Defects

This section provides an additional classification of software defects.

Bohrbug

Bohrbugs, named after the Bohr Atomic Model, are trivial defects that exhibit themselves subject to a set of conditions and can be detected easily [65]. Bohrbugs are often described as permanent defects [83] that do not change their nature with changes in the environment.

Heisenbug

Heisenbugs are defects that elude detection by changing nature through the change in execution environment, thus making their observation and correction difficult at the same time. Named after the Heisenberg's Uncertainty Principle, these defects are transient defects [83] that depend on the execution configuration and thus are often observable in a finished product, but not in its development version. A study [65] argues that all software defects are either Bohrbugs or Heisenbugs, with another [83] supporting the claim after almost two decades, using the experience over grid applications. Based on the observations made on defect types that are described later in this chapter, this research agrees with the observation that defects can be either transient or consistent (i.e., permanent), thus agreeing with the statement.

3.5.5 Other Taxonomies

Apart from the ones already described, a number of different taxonomies exist that involve code defects, that we do not describe as their types are covered one way or the other by the ones described. The taxonomies focusing on generic software defects are rare, but most of those focusing on software security include software defects in their classification scheme. A number of dissertations [72, 87] that focused on computer or network security taxonomies, included code defects either directly or through some indirect form.

Table 3.1: Properties of Analyzed Defect Taxonomies

			Bias		Properties					Scope					
#	Taxonomy / Author(s)	Focus	Subjective	Objective	#Categories	Model	Completeness	Orthogonality	Non-redundancy	Organization	System	Project	Process	Extensibility	Source
1.	HP Scheme [62]	HP	◐	●	10(21)	M	◐	◐	◐	●	◐	◐	◐	◐	[53, 54, 153]
2.	ODC [28]	IBM	◐	●	5(8)	M	◐	●	●	●	●	◐	◐	●	[53, 28, 153, 156, 114, 99, 140, 128]
3.	Errors of TEX [82]	TEX	●	○	9(15)	F	○	○	◐	○	○	○	○	○	[44, 156, 128]
4.	DeMillo & Mathur [44]	TEX	○	●	4(4)	F	●	◐	◐	○	●	●	○	○	[44, 128]
5.	Unix Security Taxonomy [6]	Unix	◐	◐	9(12)	H	○	◐	●	○	●	○	○	○	[6, 156, 162, 13]
6.	Landwehr [89]	Security	●	○	10(13)	M	◐	◐	●	○	●	○	○	◐	[89, 156, 162, 6, 12, 72, 107]
7.	Basili & Pericone [10]	Complexity	●	○	5(5)	F	○	◐	●	○	●	○	○	○	[44, 99]
8.	Mariani [116]	Components	●	○	10(18)	H	◐	●	●	○	●	○	○	○	[140]
9.	Weber [162]	Security	●	○	17(17)	H	○	◐	◐	○	●	○	○	○	[162]
10.	Leszak et al. [99]	Projects	●	◐	11(21)	H	◐	○	○	○	◐	◐	◐	◐	[140]
11.	Seaman et al. [140]	Projects	◐	◐	10(24)	H	◐	◐	◐	◐	◐	○	○	◐	[140]
12.	Nakamura et al. [120]	Projects	●	○	8(9)	H	◐	○	○	○	○	●	○	○	[120]

Legend: ●= Strong, ◐= Weak, ○= None
# Categories: Categories related to source code (Total number of categories) - counted at the most elaborate level
Model: F = Flat, M = Matrix, H = Hierarchical

3.5.6 Prominent Issues with Present Taxonomies

Table 3.1 lists a comparison of 12 existing taxonomies that treat source code defect either exclusively, or with a considerable coverage. The leftmost column lists the taxonomies, or the name of the authors in case the taxonomy is not referred by a generic name. The second column, focus, mentions the target of original taxonomy development - which is either a specific system, or specific type of projects or organizations. The Subjectivity and Objectivity columns describe the development bias of the taxonomy.

Implicit Focus to Source Code

Most defect taxonomies focus on defects in the entire system, with minimal, and often no focus, to source code defects. Despite the fact that source code contains 75% of the total defects of the software system [99], it is not the focus of an equal share of taxonomies. In Table 3.1, the column '#Categories' shows the analyzed taxonomy's focus on source code.

The taxonomies that do focus on source code (e.g., [54, 28, 6, 82, 44, 10]) suffer from other issues, as pointed out by a number of studies [44, 140], and described in subsequent sections.

Subjectivity and Objectivity

Table 3.1 shows the subjective and objective bias of taxonomies. In the table, the bias for both subjectivity and objectivity were denoted using one of the three symbols - 'strong', 'weak' and 'none'. The term 'bias' is used to express the actual methodology of developing the taxonomy - a subjective bias being the application of developer's discretion in the generation process, and an objective bias being the application of automated or automation capable rules or methods in the generation. A strong bias in any of the column indicates that the bias denoted by the column is a necessary and sufficient element in generating the taxonomy. A weak indicator denotes that the bias can be applied, but is not sufficient to generate the taxonomy on its own. A no bias indicator expresses the absence of the particular bias in the generation process. For example, a strong subjective and weak objective bias means that the developer's discretion and objective rules both can result in the taxonomy, but developer's discretion alone can generate the complete taxonomy, while the objective rules require the discretion as its supportive mean to achieve the goal.

As it is apparent from Table 3.1, most taxonomies have strong bias towards subjectivity. The taxonomies that are developed entirely on subjective discretion (e.g., [82, 10, 89]) tend to be specialized, and are often obscured for later interpretation, as can be inferred from the columns grouped under 'scope'. The human centric procedure for subjective development may also introduce more inaccuracies into the taxonomies developed in this way than their objective counterparts.

Taxonomies that focus entirely on the objective end (e.g., [44]) are likely to result in extremely concise structures that can only be used by specific software systems and groups that develop the system, as it is apparent from the 'number of categories' column. The advantage it offers over the purely subjective taxonomies is that the procedure can be applied on similar systems, but it does not create a single taxonomy that can be applied on more than one systems.

Regardless of the development process, it is argued that taxonomies are developed to serve subjective interests [90] by providing insights into the subject of interest. This argument can be extended in having taxonomies to incorporate subjectivity in their construction. Most widely adopted taxonomies (e.g., HP, ODC [28]) incorporate both the subjective and objective activities into the development process - which seems to be an inevitable quality for any taxonomy intended to be generalized over multiple subjects.

Specialization

The columns grouped under 'Scope' in Table 3.1 show the generality of the taxonomies across four perspectives - Organizations, Systems, Projects and Processes. Business entities were considered as 'Organizations' (e.g., HP, IBM or NASA), specific computer software groups or frameworks were included in 'Systems' (e.g., UNIX), specific standalone software were considered in 'Projects' (e.g., TEX) and the more general processes of development were included in 'Processes'. A taxonomy's compliance to generality across each perspective is marked as either 'strong', 'weak' or 'none'. A strong value in any of the perspectives denotes that the taxonomy is ready to be used across different entities in that perspective. A weak association is used to denote cases where a taxonomy cannot be readily applied, but can be adapted into one, across that perspective. A no association denotes that the taxonomy can neither be applied nor adapted across the perspective. For example, in row 6 of the table, Landwehr's taxonomy is reported to be applicable to different systems, with no application across organizations, processes or projects.

Most taxonomies are developed with a narrow focus for the defect targets. Often, the focus is on specific software systems [44, 155, 6, 12], specific technologies [116, 114, 115], and specific defect types, most often security faults only [162, 111, 89, 155, 6, 12, 72].

Violation of Desirable Properties

Most of the taxonomies are criticized for violating the desirable properties for effective taxonomies. Many of the taxonomies, especially the ones that are created via purely subjective discretion, are criticized for having unclear definitions, confusing mapping and overlapping categories - thus violating the orthogonality principle.

In Table 3.1, the last three properties grouped under "Properties" show the compliance to desirable properties. A strong indicator indicates that the taxonomy completely complies with the mentioned property, a weak indicator indicates the taxonomy attempts to comply with the property but contains problems that prevent its complete compliance, and a none indicator indicates the taxonomy does not attempt to comply with the property.

Most taxonomies violate the orthogonality principles by one way or the other, often through ambiguities present in their definitions, as pointed out by numerous studies [44, 6, 12, 89, 116].

The phenomena that "views to the systems have different levels of abstraction" and "the same defect may belong to different classes due to different abstraction levels" were observed as a problem to the validity of a taxonomy by Bishop and Bailey [13] but argued against by Weber [162]. Weber's argument was that, if a defect manifests this property of being classified in different groups at different abstractions, it is due to the property of the defect and not the flaw of the taxonomy.

Extensibility

Extensibility is a feature overlooked by most taxonomies, as Table 3.1 displays. In the 'Extensibility' column, a strong indicator is used to express that the taxonomy explicitly considers extensibility as one of its objectives. A weak indicator means the taxonomy does not consider extensibility as a development objective, but provides a degree of extensibility using an "Others" category or a similar construct. A none indicator indicates the taxonomy is completely non-extensible.

A question may arise at this point about our advocacy of Extensibility that we consider as a desirable property for taxonomies. The "Others" category clearly goes against the desirable property of non-redundancy while providing a degree of extensibility. The extensibility we speak in favour of, is not reflected in the extensibility provided by the "Others" category. Our argument was in favour of 'Controlled and Specific' extensibility, while the one provided by including an "Others" category in the taxonomy is both uncontrolled and unspecific. Such inclusions can create inconsistencies in the taxonomies in future, which our recommendations do not intend to. The controlled and specific extensibility that we recommend ensures the taxonomy is extended following strict rules, and under defined formal methodologies, ensuring the preservation of consistency.

3.6 A Framework for Developing Extensible Defect Taxonomies

This section describes FlexTax (FLexible EXtensible TAXonomies), a human supervised automated system for defect mapping into flexible and extensible taxonomies that are generated as the defects are being mapped.

3.6.1 Development Objectives

The development objectives of FlexTax can be summarized as,

(i) Defining objectivity of an automated system within subjective bounds of a human supervisor.

(ii) Balancing the workload between the automated system and human supervisor.

(iii) Representation of multiple points-of-view.

(iv) Ensuring flexibility and extensibility.

(v) Providing perfectly orthogonal defect mapping, instead of perfectly orthogonal definitions.

Software is ubiquitoes and it is not possible to enumerate all tasks a software performs or will perform in the future. This translates to the statement that 'all' future defects cannot be identified. Therefore, trying to develop a taxonomy that can be applied to all situations of the present and future is impractical. However, a workaround to this matter can be developed. If a taxonomy is flexible and automated, it can at least become consistent for the present situation, without a guarantee of future performance. And if it is extensible, it can provide a way to accommodate future change. The combination of flexibility and extensibility thus can make a taxonomy resilient to invalidation - although cannot guarantee its being perfect or everlasting.

Objectivity within Subjective Bounds

Quantification, classification and definition of individual defects are often subjective notions [90]. Being developed for human interpretation, it is not always possible for a taxonomy to leave off all subjective elements. On the other hand, to be precise and non-redundant, a defect taxonomy requires the employment of objective methods. In the framework, an effort was made to balance and formalize the subjectivity, instead of trying to eliminate it altogether.

FlexTax aims to define the objectivity within the bounds of subjective requirements - making the process of classification automation-possible, while retaining the flexibility of interpretation offered by subjective choices.

Workload Balance

The working principle of FlexTax aims to balance the workload between the human supervisor and automated processes. The supervisor defines the boundary of processes and specifies the directions of the processing,

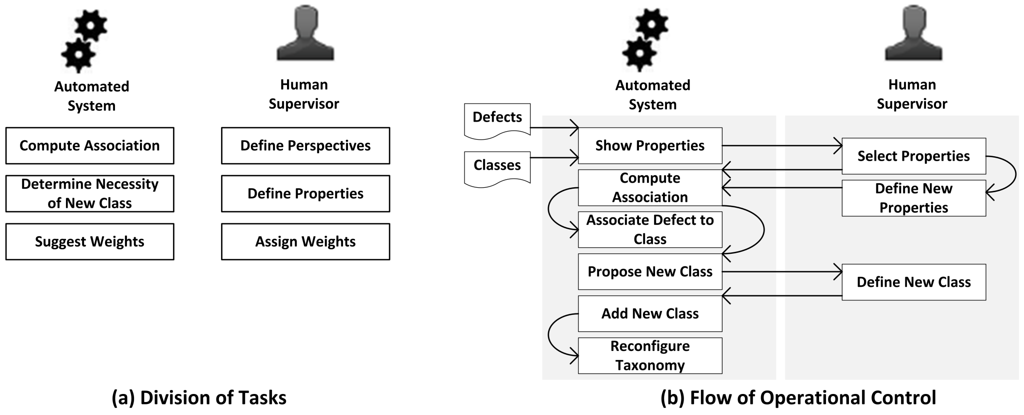


Figure 3.1: Functional Structure of FlexTax

while the automated system handles the tasks that have high effort requirements - like mapping, determining the point where a new category is required, or the reconfiguration of the taxonomy if required.

Figure 3.1 shows the concept of FlexTax. In Figure 3.1(a), specific tasks are shown for the human supervisor and the automated system and Figure 3.1(b) shows the flow of operations. As shown in Figure 3.1(a), the tasks that require high efforts, like re-configuring the entire taxonomy when a new class is added or determining the requirements for a new class, are left to the automated system while the relatively less effort heavy tasks, like approving weights and determining properties are left to the human supervisor.

FlexTax implements the development objectives through a set of perspectives, a set of attributes, one conversion function and one mapping function, as described in subsequent sections.

Multi-layered Taxonomy Representation

Different taxonomies correspond to different points-of-view and all except flat taxonomies may accommodate more than one points-of-view. Often inputs to the same classification process come from multiple and different perspectives, generating different versions of the same taxonomy. To develop a taxonomy that serves or accommodates multiple interests, a mechanism is required to represent the taxonomy in multiple directions at the same time.

To make a taxonomy beneficial through different phases of the defect reporting, detection and removal process, FlexTax considers different *perspectives* through different directions of the taxonomy. A perspective is considered as a subjective interpretation of a defect's nature from one specific point-of-view and is realized through a set of attributes that corresponds to the perspective. In other words, the attributes to defects and defect classes are chosen with a view to realize a specific perspective. A perspective is not meant for processing by the automated framework, and can be free-form.

For flat taxonomies, FlexTax uses exactly one perspective that describes the general direction of the classification. As flat taxonomies do not incorporate more than one points-of-view, one perspective suffices for its need.

For hierarchical taxonomies, multiple perspectives are used with relation to each other - each perspective being applicable to one level of classification. Every perspective except the one corresponding to the topmost level elaborates the perspective above it, and every perspective except the one corresponding to the bottommost level generalizes the perspective below it.

For matrices, FlexTax uses multiple perspectives without relation to each other. That is, every single perspective describes the taxonomy from a different point-of-view and even though one might generalize or specialize another, it is not an explicit relation.

Perspectives are realized through a set of *Attributes*, which are features describing the defects and classes. The attributes, like the perspectives, are meant for the human supervisor and do not require a strict structure (can be natural language constructs). However, the compliance of the defects and classes need to determined with respect to these attributes, and thus their definitions are not entirely free-form like those of the perspectives. An attribute definition is required to provide exactly one comparable feature for the class or defect. For example, "Assigning an entity a value larger than its maximum capability" can be an attribute, while "Assigning an improper value to an entity" is not, because the first statement can be utilized to find a strict compliance while the second cannot be as it contains the ambiguous term "improper" which might mean a number of situations.

Characterization

A defect can be characterized in many ways - and it has been done in previous research from the point of effect, root cause, or technicality. Considering the scope of the defect in this study, a single characterization approach might pose a problem with orthogonality, and may hinder the extensibility.

FlexTax considers a defect class as a quantitative characterization of a set of attributes selected in compliance with the perspectives. The collection of attributes represent a set of features determined through the specific perspectives. Any two perspectives may share attributes, but the two sets of attributes corresponding to any two perspectives are not identical. The attributes are assigned numerical weights that distinguish every combination of attributes from all other combinations. Every defect has a corresponding compliance vector that denotes its compliance to the set of attributes. This compliance vector is then converted, through a conversion function, into a point in an n-dimensional Euclidean Space that denotes the characteristics of the specific defect.

Defect classes, like defects, are represented under the same Euclidean Space and through the same mechanism. A defect class is simply an ideal defect of that class, one that complies with every single attribute relevant to that class. This approach was chosen to facilitate a few benefits. First, it classifies defect classes and defects under the same framework - making it possible to build the taxonomy from earlier taxonomies, real world defects, or a combination of both using the same techniques. Second, it provides a way to resolve the orthogonality problems with carefully chosen values for specific attributes. Third, it provides a way of prioritizing specific classes by segregating planes that contain the defect classes through any dimension.

To make classes distinct from one another, FlexTax segregates the attributes relating to a class as a set of Essential Attributes and a set of Optional Attributes, with the sets being mutually exclusive to each other. An attribute belonging to the set of optional attributes for a class can be an optional attribute for any number of other classes, but an attribute being an essential attribute to a class is exclusive to that class's essential attributes only, although it can be an optional attribute for any other class. To become a member of a class, a defect has to contain the same essential attributes, although may not have all or any of the optional attributes.

The characterization of the defect and defect class is done through a conversion function that accepts the defect and defect class's compliance vectors and computes an intermediate representation using a defined mapping function. The intermediate representation is a vector expressed in an n-dimensional Euclidean space - with each dimension expressing one perspective. Thus the intermediate representation's projection on any axis denotes the affinity of the defect or defect class towards the perspective represented by that axis. The definition of the conversion function remain fixed for any taxonomy, but are not restricted by FlexTax. Different taxonomies may have different mapping functions based on the situation under which the taxonomy is developed.

After the conversion function's characterization of a defect, a mapping function performs the mapping of a defect to a defect class under defined criteria. These criteria involve the intermediate representations and, like the conversion function, are not restricted by FlexTax, although must remain fixed for one taxonomy.

Orthogonality

A problem of orthogonality immediately arises from the procedure mentioned in the previous section. If a point corresponding to a defect (a defect-point) is equally associative to points corresponding multiple defect classes (defect-class-points), then the classification becomes undecidable. Often it is not possible to provide completely orthogonal definitions for a defect, owing to the different interpretations by different users. To account for this reality, FlexTax does not aim to eradicate all traces of non-orthogonality from a defect class's description, rather focuses on assigning the defect to the most relevant defect class should a situation with multiple associations arises.

Often, orthogonality cannot be ensured as a strict rule for taxonomies. An example is shown in Listing 3.1. A buffer overflow means accessing a piece of memory beyond the permitted range, and an off-by-one error is the offset of a desired value by one. There can be a situation where an off-by-one error results into a buffer overflow, as is depicted in the listing. Still, it is not possible to substitute the definition of one by the other because off-by-ones are not the only causes of buffer overflows, and not all off-by-one errors result into buffer overflows. If they are merged into one class, the class has to exclude situations of arithmetic errors that do not contribute to buffer overflows. To accommodate such situations, deliberate overlapping regions have to be kept in defect class definitions. FlexTax realizes this practical situation, and focuses on completely orthogonal mapping instead of completely orthogonal definitions.

Listing 3.1: Code Listing Showing Situation of Non-orthogonality

```
int a[100];
for(int i = 0; i <= 100; i++)
{
  a[i] = 0;
}
```

The fact that different types of defects correspond to different levels of response (e.g., complete failure, major deviation, minor deviation) can be characterized through prioritization of defect classes according to severity. The prioritization, in addition to expressing the relative defect severity, can also solve the problem of orthogonality to some degree by identifying the most bizarre effect resulting from a defect. FlexTax's concept of Essential Attributes assigned to classes and defects eliminate any confusing mapping scenario, where non-exclusive properties for a class are included in the optional attributes.

FlexTax considers numerical weights assigned to the defect classes - in the form of the values in the characterization vector. The defects are thus segregated into different lines (if taxonomy contains two perspectives), planes (if the taxonomy contains three perspectives) or hyperplanes (if the taxonomy contains more than three perspectives) (hereinafter, the generic term 'plane' is used to mean either of the three). Any two planes across a perspective contain two different groups of classes based on their relative importance, as set by the weight setting policy. The values are set so that the distances between planes that hold defect class points of different priorities are at least twice the maximum distance between any two defect class point on the same plane through the same perspective. The reason for this adoption is to force the defect points to be closer to the maximum significant, or maximum prioritized defect classes. Thus for confusing defects, the defect gets associated with the most severe defect class, nullifying the effect of other associations.

To implement the weight for a class, the attributes are assigned weights through a weight assignment policy. This weight assignment policy, in addition to the conversion and mapping functions, is the third configurable aspect of FlexTax that can be modified with need, but should remain constant for a single taxonomy.

Extensibility

To create extensible taxonomies, there has to be a mechanism to determine the point where new defect classes need to be incorporated into the taxonomy. In congruence with the prioritization scheme set in the previous section, it can be safely assumed that in case a defect's proximity to the nearest defect class is bigger than the distance of the prioritized planes, a new plane with higher/lower priorities needs to be introduced - that is, a new parent class needs to be introduced.

If a defect is not close enough (by means of Euclidian Distance) to any defect class under a perspective, but is not distant enough to form a new plane, a new defect class needs to be introduced under the same parent class.

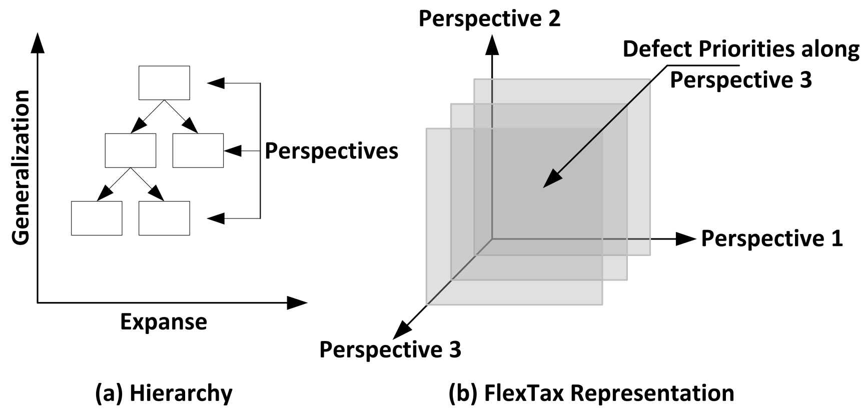


Figure 3.2: Structure and Representation of a Multi-layer Hierarchical Taxonomy

Figure 3.2 shows a conceptual view of the multi-layer taxonomy. With Figure 3.2(a) denoting the structure of a three layer hierarchical taxonomy and Figure 3.2(b) showing the representation in FlexTax. The representation shows three planes prioritized along perspective 3. A defect point would have the same priority as its closest plane.

3.6.2 Formalization

This section provides the formal framework for FlexTax. The expressions written in this section followed the conventions - using upper case letters to denote sets, upper case letters with subscripts to denote subsets, lower case letters with subscripts to denote set elements and Greek letters to denote constants.

Let,

(i) $D = \{d_1, d_2, ...\}$ a set of defects existing in code.

(ii) $P = \{p_1, p_2, ..., p_n\}$ a set of n perspectives (i.e., subjective directions) to categorize those defects.

(iii) $A = \{a_1, a_2, ..., a_m\}$ a set of m attributes for defects, determined with respect to the set P. For every $p_i \in P$, there exists a subset $A_i \subseteq A$. For $i \neq j$, A_i and A_j may not be mutually exclusive.

(iv) $W = \{w_1, w_2, ..., w_m\}$ a collection of m numerical weights corresponding to the attributes in set A. For every $a_i \in A$, there exists a $w_i \in W$.

(v) $C = \{c_1, c_2, ..., c_x\}$ is a set of defect classes, where, for any perspective p_i, there exists a $C_i \subseteq C$. For $i \neq j$, C_i and C_j are mutually exclusive.

(vi) $B = \{b_1, b_2, ..., b_m\}$ is a collection of m values to denote a single level compliance to each element $a_i \in A$. Naturally, a $B_i \subseteq B$ exists for every $A_i \subseteq A$. Unlike W and A, B has to have separate instance B_i^c for every single $c_i \in C$ and B_i^d for every single $d_i \in D$.

(vii) $M = \{m_1, m_2, m_3, ...\}$ is a collection of intermediate values for defects and defect classes. The set M can be considered a model to represent any entity, that is, any defect or defect class.

(viii) $Weight : W \rightarrow R$, where R is the set of real numbers, is a relation that assigns weights to different members in the set W.

(ix) $Convert : D \rightarrow M$ is a conversion function that converts the defect features into an intermediate representations.

(x) $Map : M \rightarrow C$ is a mapping function that maps a defect's intermediate representation to a specific defect class.

(xi) α is a predefined minimum separation between any two classes on the same plane.

Defect Characterization

For any defect d_i and defect class c_j, there is an Euclidean Vector $V_i^d = \{v_i^d(1), v_i^d(2), ...v_i^d(n)\}$ and $V_i^c = \{v_i^c(1), v_i^c(2), ...v_i^c(n)\}$ where every value in the vector corresponds to a real value determined by the association of the vector to one $p_k \in P$. The association, in this case, is denoted by the position of the point along the particular axis that represents the perspective p_k.

The values $v_i(n)$ are determined by analyzing the compliance of the defect or defect class to the set of attributes, and through association of the attributes to the set of perspectives. The conversion function accepts the compliance vector for a defect, or class, and produces the vector V_i^d or V_i^c.

The conversion function can be customized to realize different conversion policies (e.g., defining preemptive attributes, creating prioritized relations or creating attribute dependency). For this experiment, however, a simplified conversion scheme was adopted that sums up all corresponding weights towards a perspective.

$$v_i(n) = \sum_{k=1}^{m} b_k * w_k * z(k) \text{ for every } b_k \in B_i \tag{3.1}$$

where,
$z(k) = 1$ if a_k corresponds to p_n, 0 otherwise

Algorithm 3.1 shows the conversion algorithm in pseudocode.

Relating defects with classes

A mapping function decides the association between a defect and any corresponding defect class. In case a reasonable mapping cannot be defined, a new class is incorporated to the set of classes. The new class is derived from the defect that posed the requirement for the new class, and thus contains the same attributes the defect contains.

The mapping function is a configurable entity that has two distinct parts. The first part is the comparison metric, and the second part is a criterion to satisfy for the mapping. There can be a number of mapping, including average, weighted average, different distance metrics and other combinations. For the experiment to test FlexTax, a simple mapping was adopted. For the current experiment, the mapping is done through computing the Euclidean Distance between the two vectors.

Algorithm 3.1: ConvertDefectToModel

```
Data:
d: The defect to be mapped
P: The set of Perspectives
A: The set of Attributes
W: The set of Weights
$B^d$: The compliance vector for d
Result:
M(d): Interpediate Representation of d
1  Let $val(p) \leftarrow$ The value across perspective $p$;
2  for each $p \in P$ do
3      Set $val(p) \leftarrow 0$;
4      for each $a_i \in A$ do
5          if a corresponds to $p$ then
6              Set $z \leftarrow 1$;
7          end
8          else
9              Set $z \leftarrow 0$;
10         end
11         set $val(p) \leftarrow val(p) + B_i^d * w_i * z$;
12     end
13 end
14 return $val(p)$ as $M(d)$;
```

Considering $dist(V_1, V_2)$ to be the Euclidean Distance between the two vectors, that is,

$$dist(V_1, V_2) = \sqrt{\sum_{k=1}^{n} \{v_1(k) - v_2(k)\}^2} \tag{3.2}$$

and,

$$proximity(d_i, c_j) = min(dist(V_d(i), V_c(k)) \text{ for all } c_k \in C) \tag{3.3}$$

The mapping condition, for the current research, is defined as,

$$proximity(d_i, c_j) \leq T_p \tag{3.4}$$

where,

T_p = a threshold used to validate proximity.

The weights w_k assigned to the attributes have the purpose of distinguishing one defect attribute from another, and do not quantify any concrete measurement. This matter can be exploited by assigning arbitrary values for different attributes over a particular perspective.

To put the framework to use in generating a taxonomy, we have used the following rule. For all unassigned weights w_k associated with the attributes for a new defect class, if the attribute is not among essential attribute, the weight 1 is assigned. For the attributes that belong to the set of essential attributes, a total weight is distributed among them. The total weight is calculated as α more than the maximum of total attribute values of all other classes. Algorithm 3.2 shows the weight assignment policy.

Algorithm 3.2: WeightOfAttribute

```
Data:
x: The attribute to assign weight to
P: The Set of Perspectives
E: The set of Essential Properties
O: The set of Optional properties
Result:
w: Weight of x
1  Let $\alpha \leftarrow$ The minimum distance between two classes ;
2  Let $\kappa \leftarrow$ Number of elements in E ;
3  if $x \in E$ then
4      Let $p \leftarrow$ The perspective associated with x;
5      Let $v \leftarrow$ The maximum value towards $p$ for all $c \in C$ ;
6      return $\frac{v+\alpha}{\kappa}$ ;
7  end
8  else
9      return 1.
10 end
```

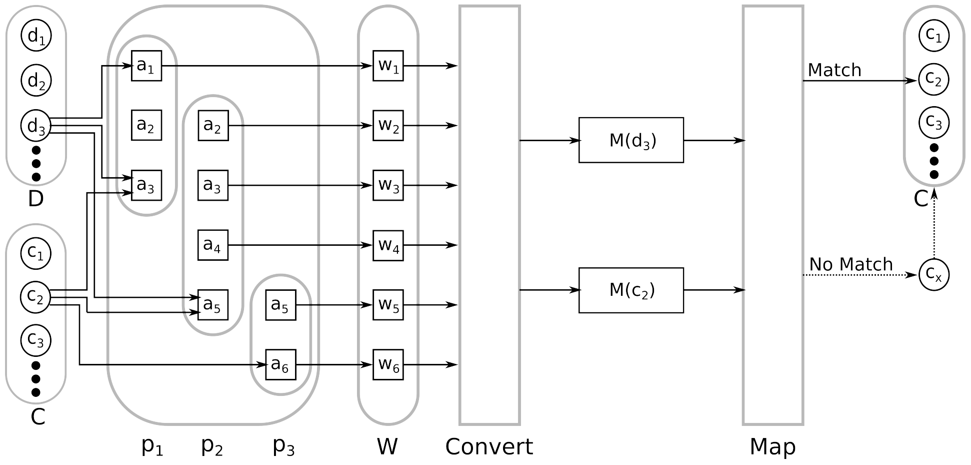


Figure 3.3: The Procedure of Mapping Defects to Classes

3.6.3 Procedure for Mapping Defects

Mapping of the defects, and building and extending the taxonomy in the process, is interactive with the requirement of feedback from the end user. In the entire process, users' feedback is used to determine the attributes and to find the affiliation to the attributes.

Figure 3.3 shows the procedure for mapping a defect. Rounded rectangles denote sets, rectangles denote functions and arrows denote the association between different artifacts. The diagram shows the set of Defects D, the set of Classes C, and six attributes a_1-a_6 distributed over three perspectives p_1, p_2, p_3. p_1, p_2 have common attributes a_2, a_3, while p_2, p_3 have common attribute a_5. The defect d_3, being associated to the attributes a_1, a_3, a_5, is converted into the model entity $M(d_3)$ by the Conversion Function and then passed on to the Mapping function. If a defect class c_2 happened to be containing the same essential attributes as d, in a previous phase, it would have been converted into the model entity $M(c_2)$ by the same procedure, and in this case, the mapping function would map the defect d_3 with the class c_2 (or any other class) based on their similarity. In case no mapping was found, FlexTax would create a new defect class, denoted as c_x that has the same attributes as d, the defect being mapped, and incorporate the class into the set of classes.

Any future defect that resembles similarity with d will then be mapped into the new class.

Algorithm 3.3: MapDefectsToClasses

```
Data: C,D,A,$B_i^d$,W,$T_p$
Result: None
for each $d \in D$ do
    if $C = \emptyset$ OR $proximity(d, C) > T_p$ then
        Let $E_d \leftarrow$ The Essential properties for d;
        Let $O_d \leftarrow$ The Optional properties for d;
        Let $X \leftarrow E_d \cup O_d$;
        Set $A \leftarrow A \cup X$;
        for each $a_i \in A$ do
            if $a_i \in X$ then
                set $(b_i \in B_i^d) \leftarrow 1$;
                set $w_i \leftarrow Weight(a_i, E_d, O_d)$;
            end
            else
                set $b_i \in B_i^d \leftarrow 0$;
            end
        end
        Designate $c \leftarrow d$;
        Set $C \leftarrow C \cup c$;
        Associate $d$ to $c$;
    end
    else
        if More than one $c \in C$ has proximity $\leq T_p$ then
            Let $c_x \leftarrow$ The one with most priorities among all matched $c \in C$;
        end
        else
            Let $c_x \leftarrow c$;
        end
        Associate $d$ to $c_x$;
    end
end
```

3.7 Case Study

This section describes a case study conducted to validate FlexTax, and a two-layer hierarchical taxonomy that was generated from that study. The study was conducted using real world defect data obtained from the Common Vulnerabilities and Exposure, or CVE, dataset [119].

3.7.1 Experimental Setup

CVE [119] is a dictionary that defines common vulnerabilities in software systems. It is not intended to be a data repository, but it provides useful defect information as a catalogue to list common vulnerabilities. We have used more than 25000 reported cases from CVE as experimental data to evaluate FlexTax.

There are multiple reasons for selecting CVE as the test repository. First, CVE contains well formed and verified descriptions of defects, and provides a defect classification of its own. Although the focus of CVE is on vulnerabilities, the originators of the vulnerabilities are defects in source code. Second, CVE contains defects from multiple systems, in contrast to the bug repositories that contain data from specific projects. Using CVE's reported instances can encompass the same dimensions as multiple bug repositories would have provided.

3.7.2 Parameter Estimation

FlexTax requires the structure of the taxonomy to be determined before the development starts, to realize its subjective dimensions. For this experiment, as the data to CVE are reported from system users / maintainers and used by developers / maintainers, a two-layer hierarchical approach (that is, one with two perspectives) was used. First perspective described the most visible effect of the defect from a professional's point of view. Second perspective described the details for the developers.

The value T_p is the threshold of proximity used to measure a defect's association with a class, and is required to be estimated before development. For this, a small experiment was conducted with three sets of 50 random defects each. The results were used to estimate initial T_p that best minimizes the incompleteness and it was found to be 50% of the predefined minimum distance between two classes, α.

64 attributes were used to classify the defects, of which 59 were used for the second perspective. For simplicity, the attributes were specified as subjective descriptions that can form a strict compliance relationship with the defect - that is, a defect either fully complies with the attribute or does not comply at all.

3.7.3 A two-Layer Hierarchical Taxonomy

The application of FlexTax over the dataset of CVE resulted into a taxonomy of 22 classes grouped under 5 groups. Following sections describe the Taxonomy. Layer-1 categories (belonging to perspective 1) are identified with an alphabetic character and the subcategories under each category (belonging to perspective 2) are marked with a number. For each category, a definition and example defects were mentioned. For clarity, these examples were chosen as simple ones for demonstrating the situation and were selected from assignments submitted by students in University of Saskatchewan's introductory programming courses[1]. The code listings provided as examples were also chosen from the same source. As an annexure to this chapter, Appendix A provides the details on the actual attributes and other parameters for developing the taxonomy.

The defects were described using three terms,

Definition 3.1. *An 'entity' is a symbolic representation of data in source code. It can be a variable, a constant, or an expression, and may denote a single value or a collection of values such as an array or a pointer to allocated memory.*

Definition 3.2. *For any entity e, the set of permitted data values* $P = \{p : p \text{ is a single value for } e\}$ *is the set of all defined values e can assume over its build environment.*

Definition 3.3. *For any entity e, the set of expected data values* $V = \{v : v \text{ is a single value for } e\}$ *is the set of values for e in its context assumed under the correct behaviour of the software. For any single entity,* $V \subseteq P$.

[1]CMPT 111 (Introduction to Computer Science and Programming), 2011 and 2012. CMPT 115 (Principles of Computer Science), 2011, 2012. Both obtained from the submitted assignments by the students and analyzed anonymously.

3.7.4 Computation (C)

Computation defects are defects that produce incorrect output due to an anomaly present in specific values of identifiers, or calculation results. FlexTax proposed three defect classes for computation defects.

Value Representation Defect (C.1)

A value representation defect is caused by the representation of values to a different set than what is expected. Such representation often truncates a range of values to a subset of itself, or generates a completely non-matching value than the entity that receives it expects. As a generic statement, a wrong value representation defect can be defined as,

Definition 3.4. *For an entity e_1 with the set of expected values V_1, a value representation defect occurs if the value of e_1 is represented to fit an entity e_2 with set of expected values V_2 such that, $V_1 \not\subseteq V_2$.*

Common examples of such defects include, but are not limited to,

(i) Type casting a value to a type able to hold only values smaller than the one being cast.

(ii) Representing a floating point value as a binary number for which the binary representation is non-terminating (not all floating point numbers fall into this category).

(iii) Type Overflows.

(iv) Integer Divisions.

(v) Assigning unsigned values to signed types (risks changing the sign).

(vi) Comparing floating point numbers as exact values.

A point to note about this defect type is, not all instances of the same statement results into a defect. Listing 3.2 shows two instances of the situations. The first instance, in line 1, is a defect because the binary representation of 0.1 is a non-terminating fraction. The set of expected values for f is, $V_f = \{x : 0.1 \leq x \leq 0.5\}$, while, due to the addition of the extra values due to the non-terminating fraction, the actual set of values will be $W_f = \{0.1, 0.2 + \delta_1, 0.3 + \delta_2, 0.4 + \delta_3, 0.5 + \delta_4\}$, making the last value out of range for the loop. However, in line 4, the defect does not exist as the values now accommodate the entire range as was defined that is, the applied values $W_f = \{0.1, 0.2 + \delta_1, 0.3 + \delta_2, 0.4 + \delta_3, 0.5 + \delta_4\}$.

Listing 3.2: Sample Code Listing Showing a C1 Defect

```
1 for(float i = 0.0; i <= 0.5; i += 0.1)
2 {
3 }
4 for(float i = 0.0; i <= 0.6; i += 0.1)
5 {
6 }
```

Value Offset Defect (C.2)

Value offset defects are introduced by a specific offset of values in any entity. These defects are usually introduced by two phenomena. First is using a constant in an operation that is off by its actual point and Second is using a wrong comparison operator introducing an off-by-one defect. In its generic form, these offsets can be positive or negative, and can be of any magnitude.

Definition 3.5. *For an entity e with set of expected values V and set of permitted values P, a value offset defect occurs if, for determinable and defined values w for e, it holds that* $\exists w : w \in P, w \notin V$.

Examples of the defect includes, but are not limited to,

(i) Considering the minimum array index as 1 while it is actually 0 (and vice versa).

(ii) Considering the maximum array index as the number of elements of the array, while it is one less (and vice versa).

(iii) When counting number of elements in a range, not considering both of the terminal entities.

Listing 3.3: Sample Code Listing Showing a C2 Defect

```
#define SIZE 100
int a[SIZE];
for(i = 0; i <= 100; i++)
{
  a[i] = 0;
}
```

The Code Listing 3.3 shows a common off-by-one error as the limit of the for loop is one step more than it should have been. Other varieties of the value offset defects occur in estimating a value and in recurrent structures.

Undefined Outcome (C.3)

This class includes defects that result into undefined states or undefined results from operations performed on them. Undefined states can originate by using an entity whose initial states are not known and no determinable operation has been performed on it before its use as an r-value, or by using an entity that had been subjected to an operation for which the outcome is not known. The generic form of this defect can be described as,

Definition 3.6. *For any entity e with set of expected values V and set of permitted values P, an undefined outcome occurs if for applied values w of e it holds that* $\exists w : w \in P$ *but* $w \in V$ *is undecidable*

Common examples of this defect includes, but are not limited to,

(i) Division by zero.

(ii) Using an uninitialized value.

(iii) Using an entity that suffered from Defect C1 in an operation.

Listing 3.4: Sample Code Listing Showing a C3 Defect

```
int Sumup(int n)
{
  int sum;
  for(int i = 0; i <= n; i++)
  {
    sum = sum + i;
  }
  return sum;
}
```

Listing 3.4 shows a situation where the use of an uninitialized value results into the defect. The value returned in this case cannot be determined due to the initial undetermined value in sum, and the outcome of the defect will propagate through any statement and scope this function's return value is used in.

3.7.5 Logic (L)

Logical defects are defects that arise from logical constructs and control flow. FlexTax proposed eight logical defect classes.

Improper Checks (L.1)

This class includes the defects where the omission or an error in the check for an entity results into an improper validation of any data. The defect can exhibit itself as either an absence of a validation, or by using a condition as a validation artifact that has either improper operator or operand. In generic form, it can be defined as,

Definition 3.7. *If a validation statement allows a set of values W for an entity e with expected set of values V and permitted set of values P where it holds that, $\exists w : w \in W, w \notin V$ OR $w \notin P$, then the validation statement suffers from an improper check.*

Common examples of the Improper Checks include, but are not limited to,

(i) Not providing a check in a necessary place.

(ii) Creating a check with wrong, but relevant, operator (i.e., the operator does not create a compile error as it is relevant, but introduces a flaw due to being incorrect).

(iii) Creating a check with wrong, but relevant, operand (i.e., the operand does not create a compile error as it is relevant, but introduces a flaw due to being incorrect).

(iv) The C2 Defect occurring in the condition part of a validation statement.

(v) Using an irrelevant operator (e.g., = in case of ==).

Listing 3.5: Sample Code Listing Showing an L1 Defect

```
int function(void)
{
  int value, checking_value;
  ...
  if(value = checking_value)
  {
    ...
  }
  else
  {
    ...
  }
}
```

The code fragment presented in Listing 3.5 shows a common defect drawn from the submitted assignments in one of the undergraduate introductory programming course in the University of Saskatchewan. The check results into a tautology as assignment operators always return a reference to the object assigned to, which can only be interpreted as a true value.

Improper Terminal Condition (L.2)

This class includes the defects that are introduced in case of recurrent structures by improperly estimating the terminating condition. The defect can exhibit itself as an infinite loop or recursion, as a loop or recursion that terminates early or fails to terminate at the proper point.

A difference with the previous class (L1) is that, the L1 defects result in wrong outcome or execution path, but does not involve controlling recurrent structures. In its generic form, this defect can be described as,

Definition 3.8. *For any recurrent structure control entity e with the set of expected values V and the set of permitted values P, an improper terminal condition occurs if for w as values assigned to e, $\exists w : w \notin V$ holds.*

Prominent defect types under this class include, but are not limited to,

(i) Not using any terminal condition.

(ii) Exhibiting a C1, C2, C3, or L1 defect in the terminal condition.

(iii) Making a terminal condition an tautology or a contradiction.

(iv) Not updating a loop controller inside the loop.

Wrong Operation (L.3)

This class includes defects that involve doing one operation where another was appropriate. Such action either makes the value deviate from the expected set of values or, in case of repetitive code, violates the expected relation between the consecutive values inside the set of expected values. In its generic form, this defect can be described as,

Definition 3.9. *For any entity e that is a resultant from any operation with the set of expected values V, the operation is a wrong operation if any value of e does not comply with the values $v \in V$ or, for any value $v_i, v_j \in V$, the relation $v_i \rightarrow v_j$ is violated for $j = i \pm 1$.*

Listing 3.6: Sample Code Listing Showing an L3 Defect

```
for(i = 0; i < 10; i--)
{
   ...
}
```

Prominent defect types for this class include, but are not limited to,

(i) Using = in place of == and vice-versa.

(ii) Using < in place of > and vice-versa.

(iii) Using ++ in place of −− and vice-versa.

Flaws in Algorithm (L.4)

This category includes the defects that are originated from a flaw in the algorithm but are traceable from source code. There can be many different defect types constituting to this class, and not all can be listed (due to the infinite possibilities of algorithm design). Often the comprehension of this defect requires design knowledge for its comprehension.

Definition 3.10. *For any entity e with the set of expected values V, if the relation $v_i \rightarrow v_j$ between any two values of V, v_i and v_j becomes different that what is expected, it denotes a flaw in algorithm.*

The flaws in algorithm is not strictly a source code defect (although it is often traceable from source code) because detecting such defects require the knowledge of the algorithm or design in addition to the information inferred from source code.

Prominent defect types in this class include, but are not limited to,

(i) Not saving a value when required.

(ii) Not updating a value when required.

(iii) Not setting a value when required.

(iv) Any of the other defects if introduced by the algorithm.

Performance Issues (L.5)

This class includes defects that do not cause a wrong output or deviated behaviour, but could be implemented in a better way. An example is running a loop more than it should be run (not an Improper Terminal Condition, as it doesn't violate the expected values). A classic instance of the defect is a brute-force prime number checker that checks every factor of a number to determine if it is prime while it suffices checking only the factors that are as large as the square root of the number[2].

Definition 3.11. *For any entity e with the set of expected values V, if the same set V can be, but is not, achieved with a mechanism less burdensome in terms of resource usage, execution time, or any other performance measure, then the present implementation denotes a performance defect for e.*

Performance issues can be caused by a number of faults in the program, starting from benign issues like keeping unused variables, using redundant variables or using extra complex logic. For the most part, detection of performance issues requires knowledge of the architecture and design of the software and thus these defects are not strictly source code defects.

Prominent defect types in this class include, but are not limited to,

(i) Declaring never-used entities.

(ii) Allocating more than required.

(iii) Not using part of the allocated memory.

(iv) Making more than one conditional branch to do the same task.

Improper Exception Handling (L.6)

This class includes defects that result from exception handling problems. Three situations may be considered as problems with exception handling, the first is handling an exception for a code block that does not generate any exception under any condition, second is not handling an exception for a code block that might generate an exception, and the third is handling a wrong exception for a code that generates an exception. In its generic form, this defect can be described as,

[2]This particular performance issue exists in case of the brute-force algorithm only, and does not affect the advanced or efficient algorithms.

Definition 3.12. *For any block of code, if the block is able to generate a set of exceptions E_G and the exception handling code considers the set of exception E_H where $E_H \neq E_G$, the situation denotes an improper exception handling defect.*

Prominent defect types in this class include, but are not limited to,

(i) Not catching an exception.

(ii) Catching a wrong exception.

(iii) Catching an exception, but not handling it.

(iv) Catching the right exception, but handling it wrong.

Control Flow Error (L.7)

The control flow error for any part of the code is a piece of code that results into an improper control flow. Examples include making a condition a tautology or a contradiction or creating a branch of code that will never be reached. The difference between the Improper Checks and the Control Flow Error is that, unlike Improper Checks, Control Flow Errors use valid constructs, but the semantics are invalid for the checking, while the Improper Checks defect involves operators that can be detected from the compiler's perspectives.

Definition 3.13. *If, for a validation statement, the outcome has a set of expected values V, while the actual values that are produced as the outcome has a set W, and $W \neq V$, then the statement suffers from a Control Flow Error.*

Prominent defect types in this class include, but are not limited to,

(i) Making an always-taken branch.

(ii) Making a never-taken branch.

(iii) Creating unreachable code.

Design Non-conformance (L.8)

This category includes defects that are resulted from the difference from design. These defects differ with the other logical defects in that, they do not always result into non-functioning code, or in any code that in itself can indicate the problem. The examples can be an improper module interface definitions where the module is functioning and can be used, but not in the way the design document specifies - creating problems in other parts of the software implementation. This type of defects are not present entirely on the source code.

Prominent defect types in this class include, but are not limited to,

(i) Replacing efficient sorting with inefficient ones.

(ii) Changing design data structures

(iii) Introducing new code.

(iv) Not implementing part of the design code.

Memory (M)

Memory related defects are defects that involve the access or manipulation of the computer's memory. Flex-Tax proposed four defect classes for memory defects.

Invalid Memory Reference (M.1)

Invalid memory references occur when a piece of non-existing memory is referred in either a read or write operation, or a read-only memory is accessed for a write operation. Examples include referring to unallocated memory or referring to a memory that has been deallocated. Often exhibited as a null pointer dereference, this defect can be caused by the undefined outcome defect on a memory entity.

Definition 3.14. *For an atomic or collective entity e denoting a memory region for use with the set of permitted values P and the set of Expected values V, if, for applied values w for e, it holds that, $\exists w : w \notin P$ OR $w \notin V$, the operation denotes an invalid memory reference.*

Listing 3.7: Sample Code Listing Showing an M1 Defect

```
void function(void)
{
  char *p;
  cin >> p;
  return;
}
```

In the code fragment shown in Listing 3.7, the pointer p contains a garbage value that, while trying to be accessed, will result into a segmentation fault. In case p was initialized to 0, it would have caused a null pointer dereference.

Prominent defect types in this class include, but are not limited to,

(i) Null-pointer dereference.

(ii) Invalid pointer dereference.

(iii) Using an incompatible value as a pointer.

(iv) Creating a C1 or C2 or C3 defect in a memory entity.

Improper Deallocation (M.2)

An improper deallocation occurs when a piece of memory is tried to be deallocated in the wrong way. This defect can be introduced by a number of mechanisms, including, deallocating a non-existing memory, deallocating an already deallocated memory, trying to deallocate a null pointer or deallocating a memory with a wrong operator (malloc() vs. delete, new vs. free()).

Definition 3.15. *For an entity e denoting a memory region for use with the set of expected values V and set of permitted values P, an improper deallocation occurs if $P = \emptyset$ or a value w is deallocated for e where $w \notin P$.*

Listing 3.8: Sample Code Listing Showing an M2 Defect

```
void function(void)
{
  char *p = new char[SIZE];
  ...
  delete[] p;
  ...
  delete[] p;
  return;
}
```

Prominent defect types in this class include, but are not limited to,

(i) Deallocating deallocated memory.

(ii) Deallocating unallocated memory.

(iii) Using wrong deallocation procedure (new $->$ free(), malloc() $->$ delete, new[] $->$ delete, new $->$ delete[])

Memory Leaks (M.3)

This category includes defects that result from not releasing an allocation memory after the operations finishes. Ideally, if M is a set of allocated memory for an operation, $M = \emptyset$ should hold when operation finishes.

Definition 3.16. *For an entity e denoting a memory region for use with the set of permitted values P and a set of expected values V, a memory leak occurs if, after deallocation, $P \neq \emptyset$ OR $V \neq \emptyset$.*

Prominent defect types in this class include, but are not limited to,

1. Not deallocating an allocated memory.

2. Deallocating only part of an allocated memory.

Over/Underflow (M.4)

A memory overflow occurs when a piece of memory is accessed with more data than it can hold, or is accessed beyond its maximum limit. An underflow is the same condition occurring beyond the minimum limit of the memory.

Definition 3.17. *For an entity e denoting a memory region for use with the set of expected values V and the set of permitted values P, if for an applied value w it holds that, $\exists w : w \notin V$ but w is adjacent to V, the operation denotes a memory over/underflow.*

Prominent defect types in this class include, but are not limited to,

(i) Accessing an allocated memory beyond the last valid address.

(ii) Accessing an allocated memory before the first valid address.

(iii) Trying to write data bigger than the size of allocated memory.

(iv) Not terminating the allocated memory with right marker.

3.7.6 Data, Interface and Input/Output(D)

This group of defect classes contains defects that arise from data usage, interface specification and usage and the input/output to the software or a specific module. FlexTax proposed five defect classes under this group.

Interface Mismatch (D.1)

This defect type denotes a mismatch between the interfaces of a system. The difference is often an architectural or design issue and are not usually traceable from source code. The ones that can be traced from source code exhibit discrepancies in either parameter numbers, types, values or ordering for function usage.

Definition 3.18. *An interface mismatch occurs where, for any invocation of an interface that require the set of entities E_R the actual used set of entities is E_A and $E_R \neq E_A$*

Prominent defect types in this class include, but are not limited to,

(i) Calling a function with wrong parameter types.

(ii) Calling a function with repeated parameters.

(iii) Using default values for parameters.

(iv) Creating a C1 defect in the parameter of a function.

Data Mismatch (D.2)

A data mismatch is the use of data values not proper for the use in an interface. The difference between interface mismatch and data mismatch is that, interface mismatch uses wrong entities for an interface, but data mismatch uses right entities, with wrong states.

Definition 3.19. *A data mismatch occurs when, for an entity e_1 required to invoke an interface, the set of expected values as used in the interface is V_1, but the interface is invoked with an entity e_2 with the set of expected values V_2 for which it holds that, $V_2 \not\subseteq V_1$.*

Prominent defect types in this class include, but are not limited to,

(i) Calling a function with right parameter types, but wrong values.

(ii) Creating C3 defects in the parameter of a function.

(iii) Casting pointers / values to fit function call.

Improper Input Validation (D.3)

This defect occurs from not validating the inputs to a module or program before they are processed. This defect is actually an L1 defect applied on the input to a module or the entire software.

Definition 3.20. *An improper input validation defect occurs if, for a module or program with the set of input entities E, there exists an L1 defect for any $e \in E$.*

Prominent defect types in this class include, but are not limited to,

(i) Not validating an input before use.

(ii) Using an invalid validation for an input.

Missing or Extra Input (D.4)

A defect in this category occurs if the software or any module is put to work with insufficient or extra input required for processing. A missing input can cause the software to crash, while an extra input may introduce an error in computation.

Definition 3.21. *A missing or extra input defect occurs where, for any invocation of an interface that require the set of entities E_R the actual used set of entities is E_A and $|E_R| \neq |E_A|$*

Prominent defect types in this class include, but are not limited to,

(i) Not providing enough inputs.

(ii) Missing fields in composite entities used as input.

(iii) Providing extra inputs for variable list modules.

(iv) Providing extra fields in composite entities used as input.

Improper Abstraction (D.5)

This defect arises from providing improper access to a data member of a module. The term improper access means either to provide access to a data member that should not be accessed, or to restrict access to one that should be accessed.

Prominent defect types in this class include, but are not limited to,

(i) Failing to provide access to a member.

(ii) Passing an entity by-reference that should be passed by value and vice-versa.

(iii) Returning reference to a private/protected entity by a public function.

(iv) Not providing an interface to interact with a private data member.

3.7.7 Synchronization (S)

Synchronization defects are defects that occur in a multiprocess / multithread software and involve the sequencing or interleaving of the processes or threads. FlexTax proposed two defect classes under this group.

Prohibitive States (S.1)

This category contains the defects that result into a prohibition on any block of code into completing its task. Examples include deadlocks, process starvation and improper priority assignments.

Definition 3.22. *A prohibitive state occurs if the software's execution reaches a point where no further execution is possible until a certain condition is satisfied, and the condition cannot be satisfied immediately under the current situations.*

Prominent defect types in this class include, but are not limited to,

(i) Deadlock.

(ii) Data Race.

(iii) Process Starvation.

Improper Sequencing (S.2)

This category includes the defects that results into effects coming from improper sequence of execution for different blocks of code. Examples include atomicity violation, improper serialization and lock-and-release problems.

Definition 3.23. *An improper sequencing defect occurs if, through the interactions among multiple entities or processes, the order of execution impels the software to enter into undesirable states which could not have occurred had the sequencing been different.*

Prominent defect types in this class include, but are not limited to,

(i) Atomicity Violation.

(ii) Unsynchronized update.

(iii) No Locking mechanism for critical objects.

3.8 Evaluation of the Taxonomy

This section describes the evaluation of the taxonomy generated by FlexTax to verify its effectiveness. FlexTax cannot be directly evaluated as it is a framework and it possibly is the only one of its kind. The taxonomy was evaluated to point out its strengths and weaknesses which, in turn, would assess FlexTax.

3.8.1 Evaluation Directions

The two-layer taxonomy developed using FlexTax has been evaluated from three perspectives

(i) Completeness: There must not be any defect unaccounted for.

(ii) Non-redundancy: There must not be a defect class with no defect mapped in it.

(iii) Orthogonality: There must not be a confusion in mapping defects to classes.

For a comprehensive evaluation, two different evaluations were done.

(i) Verification using Machine Learning Classifiers

(ii) Manual Cross-matching with established defect databases

Table 3.2: Results of the 10-fold Cross Validation

		Pass										
#	Classifier	1	2	3	4	5	6	7	8	9	10	Avg.
1.	J48(C4.5)	0.92	1.0	0.98	0.95	0.99	0.97	1.0	0.99	0.94	0.94	0.968
2.	Naive Bayes	0.97	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.996
3.	Bayes Net	0.99	1.0	0.98	0.97	1.0	0.97	1.0	1.0	0.99	0.99	0.989

3.8.2 Verification Using Machine Learning Classifiers

FlexTax uses defect features in the form of Attributes to specify the defects and classes, and the defect feature comparison as the classification metrics to determine the defect's affiliation to a class. When it is used for generating a hierarchical taxonomy (as was done for the current research), the approach becomes an instance of classification using information gain (as new classes are added when there is significantly different information to be obtained from them). This quality makes the taxonomy an appealing candidate for a machine learning algorithm that works on information gain for classification. It was exploited in our endeavour to verify the accuracy of FlexTax with a J48 machine learning classifier using the Weka Data Mining Software [66].

A random sample of 1000 classified defect instances, along with their feature vectors (the compliance vector, B) were chosen from the defects classified by FlexTax. The Weka Toolkit [66] was used to train a J48 Classifier (a variant of the C4.5 machine learning classifier) and to apply it on a 10-fold cross validation over this 1000 defect set. The results are presented in Table 3.2. The classifier was chosen for its explicit treatment on information gain induced classification.

To further validate the case study, the same experiment was repeated under the Naive Bayes and Bayes Net classifiers, with results presented in the same table.

As can be observed from the data, for 96.8% cases under the J48 classifier, the defects were mapped exactly as expected. Of the remaining 32 defects that did not match, 29 were integer overflows (C1) mapped to memory overflows (M4) - due to the similarity in their features. The other three involved confusing scenarios that FlexTax resolved using prioritization, but J48 (C4.5) failed to handle as there was no equivalent scheme for the classifier.

Both the the Naive Bayes and Bayes Net classifiers reported close to 100% accuracy, owing to the fact that even though orthogonality problems may exist between defects, their feature sets are essentially independent. However, we would like to consider the J48 classification as the most accurate one due to the particular nature of the classifier.

The results from classification demonstrate that the classification mechanism for FlexTax is accurate and feasible. The same machine learning classifiers could be used as the main classification technique for FlexTax, but it could pose a problem in resolving orthogonality issues, or could alter the structure of the already established taxonomy, against both of which FlexTax has its own specific mechanisms. FlexTax uses prioritization and proximity calculation to balance such scenarios. Prioritization helps to balance the orthogonality problems, while proximity calculation preserves the consistency of the taxonomy.

Established Defect Databases

CVE is a dataset that defines defects leading to vulnerabilities for software systems and have been used by many organizations. Among the over 50000 entries in CVE from multiple systems, more than 25000 belong to source code defects and were processed. Rest of the entries, including repitition, fall under policy and architecture related issues that fall beyond the scope of FlexTax. The classification generated the results as presented in Figure 3.4 under 'Set 1'. In the figure, classes C, L, M, D and S denotes the collective (i.e., Perspective 1) Computation, Logic, Memory, Data and Synchronization defects with respect to the total defects. The classes C1-C3, L1-L8, M1-M4, D1-D5, S1 and S2 denotes the specific classes belonging to Perspective 2 and the corresponding bars show the defect distribution within the same group of classes (not with respect to total defects). As a secondary set, the defect dataset recorded from the projects implemented under this research was used. The secondary set, denoted as 'Set 2' in Figure 3.4, contains 208 defects from general software development tasks. Observations on the data are reported in the next section.

Unfortunately, due to the subjective variance incorporated in taxonomies, there is no standard for a correct taxonomy that can be used as a benchmark. CVE provides its own taxonomy comprising of 41 classes of which 30 relate to source code defects, but the focus of CVE on the impact of the defect, and not the defect itself - making it improper for exact one-to-one comparison of defects or classes. However, a case-by-case comparison can be done if the defect features can be inferred from CVEs categories. From the defects classified by FlexTax, a random sample of 5% was chosen and the defect class attributes were listed. For each defect, the attributes of the corresponding CVE category, where they can be inferred, were listed. The two sets of attributes were cross matched. It was found that for all except 14 defects, the essential defect class attributes for FlexTax's generated taxonomy were supersets of the corresponding CVE attributes - establishing the point that should this classification scheme were used for CVE instead of the handpicked classification applied at present, the defects would have belonged to the same class as they do now.

The 14 defects that showed discrepancy between the generated taxonomy and the corresponding category in CVE were analyzed more closely. Their classification by FlexTax was found to be more accurate than the CVE classifications. The reason of the discrepancy was that, FlexTax considered all properties the defects exhibited while CVE considered only the most severe properties. As FlexTax considered all properties, their highest affiliation to a class became the class they are associated in. And as CVE considered only the most severe properties, the most severe outcome they cause became their class of affiliation. Our analysis revealed that, often a defect can cause a single most severe problem to be classified into one category, but the collective influence of a lot of other not-so-severe impacts can outweigh the severity of the single incident.

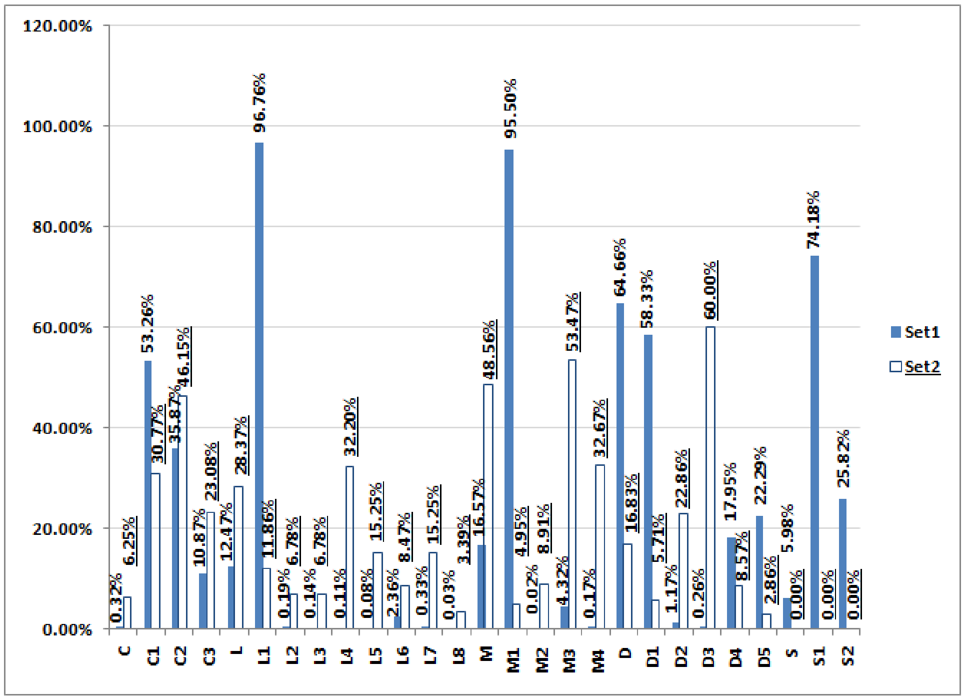


Figure 3.4: Distribution of Defects in the Generated Taxonomy

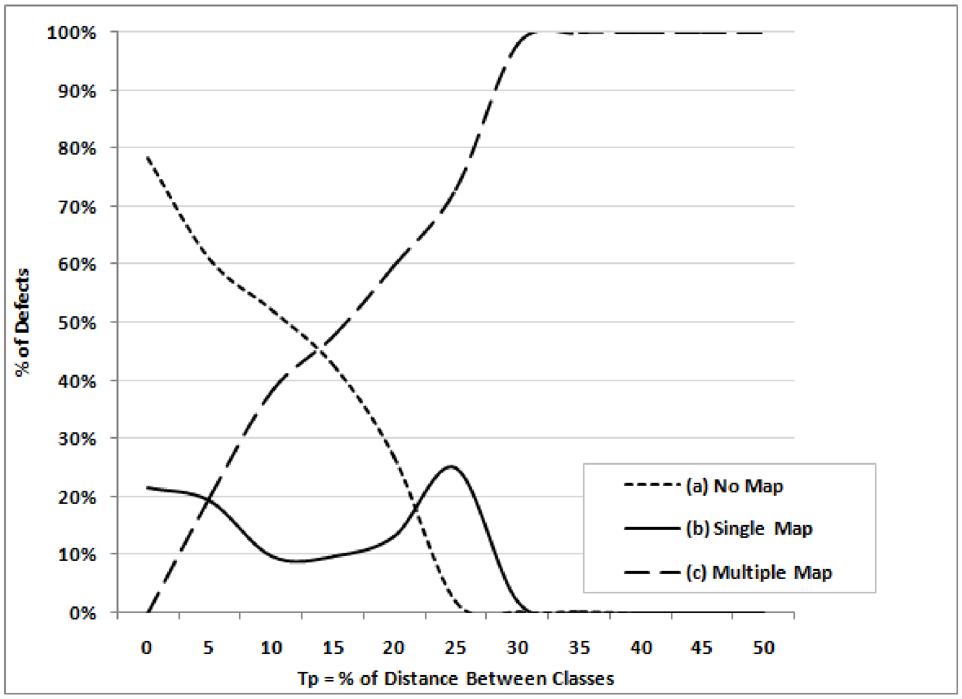


Figure 3.5: Unmapped and Multiple Mapped Defects with Differing Values of T_p

Estimating the value of T_p

The threshold T_p poses a two sided problem. Evidently, it can be deducted that with a sufficiently large T_p, only one defect class is required. This generalizes into the statement that, the higher the value of T_p, the lower is the number of defect classes. On the other hand, having a higher T_p will also give rise to the chances of mapping the same defect to multiple classes. That is, the lower is the T_p, the higher is the chance of orthogonality.

To test a suitable value, instead of repeating the experiment over such a large dataset, we have evaluated different T_p against the generated taxonomy. The results are presented in Figure 3.5. In the figure, three trends show (a) the defects having no class with distance less than T_p, (b) the defects having exactly one class within the bounds of T_p, and (c) the defects having multiple classes within the bounds of T_p.

As apparent from Figure 3.5, the value of T_p as 12-15% of the average distance between any two consecutive defect classes is the best balance between the two. However, as orthogonality issues are not sure to occur in case of multiple match, but incompleteness is a certain phenomenon in case any defect is left un-mapped, we would recommend to use a value of T_p that minimizes the number of non-matched defects. In case of the developed taxonomy a value of 50% was used.

A point to mention is that, having multiple distances to classes less than the threshold does not definitely identify orthogonality problems. Some of the cases will result into orthogonality issues if the minimum distance is the same to more than one classes. In our taxonomy, no orthogonality issues arose.

3.8.3 Observations and Justifications

(1) In Figure 3.4 (Set 1) the classes C, M, L, D, S encompass a full range of defects (i.e., 100%) and the calculation was made compared to total defects. This states the fact that no defect was left out in the classification, proving the completeness of the generated taxonomy. Same observation is made with Set 2.

(2) In Figure 3.4 (Set 1), some categories have very low distributions, but none effectively reaches 0, the lowest defect density translates to 1 defect in the total set. A question may arise about the requirement of such categories. In reality, this is a strength of FlexTax to recognize categories like these. FlexTax incorporated the category only if the feature set of the first defect mapped to this category was distinct enough to demand a new category. Had it not been mapped with the low density category, the defect would have mapped to a wrong category in consideration of its features. Set 2 further strengthens the requirement of these categories, often showing considerable population in categories that have low population over Set 1. Set 2 has a few categories with 0 distribution - attributed to the fact that the taxonomy was not developed using Set 2.

(3) In Figure 3.4 (Set 1), most distribution is in the data related defects, as opposed to the memory defects as described by CVE [119] in its own classification. The reason behind it is that, CVE, being focused on vulnerabilities, considers the defect's impacts where FlexTax considered the underlying causes. Most of the vulnerabilities classified by CVE in the memory defects were originated by using unauthorized access or using wrong parameters, both of which fall under data related defects in FlexTax's classification.

(4) In Figure 3.4 (Set 1), most defect concentration for any group is in the first class of the group. This can be explained by a peculiarity of FlexTax. As the classes are created from human identified features, naturally most of the significant features are associated with the first class, and unless they are later reassigned to other classes, they remain associated with the first class, drawing most defects into the area of distribution. Set 2 does not seem to correspond to this observation. As Set 2's features were not used in developing the taxonomy, it did not bias the features to be concentrated in the first category.

(5) In Figure 3.5, the Trends (a) and (c) exhibit a reciprocal relation, as is expected from the role of T_p in the generation process. However, Trend (b) should decrease uniformly as T_p increases, which it does in general, with an unexpected rise at $T_p = 10\%$ to 25%. A closer look discovered that some defect classes had too many optional attributes that, by cumulative weight, became almost equal to the essential attributes to other classes that had less weight for essential attributes. Defects containing one class's essential attributes and many of another's optionals tend to get a high affinity to both. This problem can be solved by making classes further apart so that the optional attributes do not gain enough weight to be equal to the essential ones of any class (i.e., it can be resolved by using a larger α).

(6) In Figure 3.5, when $T_p = 0\%$, Trend (b) should be 100% and Trend (a) should be 0%. It did not happen in this case because T_p was varied against a fixed set of classes and defects, instead of an open set with as many classes as there can be. But the 80% for Trend (a) at $T_p = 0$ indicates that in case $T_p = 0\%$, there would have been 20% more classes than there already are (not equal to total defect count because same defect is reported in different instances). Therefore, for a handpicked classification, one can expect around 27 classes considering the 20% increment. CVE's own handpicked classification system lists 41 defect classes, of which 11 do not belong to source code defects (and defects belonging to those classes were excluded from all analyses), bringing the classes close to the number we predicted.

(7) As apparent from Figure 3.5, the value of T_p as 10%-15% of α is the best balance between the multiple and zero mapping situations. However, as orthogonality issues are not sure to occur in case of multiple match, but incompleteness is a certain phenomenon in case any defect is left un-mapped, we would recommend to use a value of T_p that minimizes the number of non-matched defects, which, in this case, is around 30% of α.

(8) Based on the previous observations, if we compare the generated taxonomy with Table 3.1, for all of the last three columns under 'Properties', FlexTax will score a strong compliance, as it is strongly orthogonal, complete and non-redundant. Same observation will be repeated for 'Scope's. Also, in the 'Extensibility' column, FlexTax will be a strong candidate. Hence, In comparison to taxonomies presented in Table 3.1, FlexTax will provide the maximum benefit to all taxonomies considered, close to the ODC, but outperforming ODC in both completeness and flexibility.

Table 3.3: Comparison of Effort Requirements for FlexTax with Handpicked and Automated Taxonomies

Non-Effort-Intensive Tasks				Effort-Intensive Tasks			
	Responsible				Responsible		
# Task	FlexTax	Handpicked	Automated	# Task	FlexTax	Handpicked	Automated
1. Fixing Guidelines	User	User	User	5. Deciding New Class Addition	System	User	System
2. Creating Classes	System	User	System	6. Defect Mapping	System	User	System
3. Determining Class Attributes	User	User	System	7. Reconfiguration	System	User	User
4. Determining Defect Attributes	User	User	User				

3.8.4 Improvement on Human Effort

For a taxonomy generated manually, a human supervisor has to design the classes, map the defects, and carry on any reconfiguration activity on the entire taxonomy should the need arise in any case.

FlexTax removes the burden from the human supervisor to calculate and cross-match the affinity of defects to the already existing classes, and to determine the point for new class addition. FlexTax also provides a set of base attributes as obtained from the specific defect that created the need for the new class - thus relieving the supervisor from the work of figuring them out.

Comparison of FlexTax's working procedures with handpicked and automated taxonomies are presented in Table 3.3 with columns 'Handpicked' and 'Automated' denoting the two cases. Comparing with handpicked taxonomies, three of the most effort-worthy tasks, decision on new class addition, defect mapping and reconfiguration is left to the system in FlexTax, relieving the user of the burden of a large workload.

Automated taxonomies achieve almost the same improvement in terms of effort, with two major differences. The determination of class attributes requires subjective supervision over objective choices, which is ignored by the automated taxonomies. Additionally, reconfiguration in an automated taxonomy often requires the regeneration of the entire generation process, as is the case in DeMillo and Mathud's taxonomy [44] - in which FlexTax has a clear advantage of 'reconfiguring without regeneration'.

3.9 Defect Similarity

According to the best of the knowledge of the author, no research has so far defined defect similarity from defect class descriptions. As most of the taxonomies are handpicked and depend entirely on subjective discretion of the developer, it is not often feasible to define the similarities on their structures.

As FlexTax, despite incorporating subjectivity in its generation process, defines defects under strict mathematical constructs, it is possible to establish multiple measures of similarity among the defect classes. In this section, an attempt was made to compare the classes of the generated taxonomy with each other under different similarity metrics.

In describing a defect similarity, two notions have been used. A *Strong Similarity* between two defect classes c_i and c_j indicates that the classes c_i and c_j share similar features in a way that any defect belonging to c_i would certainly have belonged to c_j should c_i be non-existent, although the opposite might not be true. A *Weak Similarity* between the classes indicates that c_i and c_j contain common features or properties, but are not so strongly coupled that in the absence of one, the defects belonging to it would get mapped to the other.

This statement of similarity defines the underlying principle of defect similarities, but leaves the exact similarity measures open for a variety of adaptations. Considering the vast expanse of the feature comparison field, it can be assumed that there can be at least as many comparison metrics as there are techniques to represent the features, which makes the count very large (for example, a recent paper [30] lists 76 comparison techniques on binary feature vector only). If the customized techniques (like the one adopted for FlexTax) are added with it, the options become even larger.

In the generated taxonomy, the classes in Perspective 1 (C, L, M, D, S) are generalizations of their underlying classes (C1-3, L1-8, M1-4, D1-5, S1-2). They are designed to encompass the features of the Perspective 2 classes that fall under them. Therefore, in designating similarity, the Perspective 1 classes do not provide any value as they are meant and designed similar to the Perspective 2 classes. To make it relevant, only Perspective 2 classes were included in the similarity measurement study.

3.9.1 Similarity Coefficients

A similarity coefficient, or similarity score, expresses a quantitative characterization of the similarity between two artifacts of interest. In the context of FlexTax, the similarity measure is a numerical score that identifies a defect class's similarity with another.

To describe this study, the similarity scores were normalized in the range [0, 1]. The normalization procedure was adopted for easier representation of the outcome from different similarity coefficients under the same framework (a graph, in this case). The particular range of [0, 1] was chosen for its direct association to representing percentage values (which was used as the coefficient for one case).

A point to mention is, a defect class's similarity score to another is the maximum (1.0) does not imply the two classes are the same, or that they can replace one another. It only specifies that the degree of similarity in the two class' design and perception is the maximum as it can be under the current context.

It is expected that defect class similarity expressions shall be able to describe a few properties for the defect classes. First, as FlexTax explicitly incorporates both subjectivity and objectivity, defect class similarity measures should be able to cover both dimensions. Second, the similarity measures should be able to predict how the defects will be redistributed in case of a change in the taxonomy. Finally, the similarity measures are required to express the semantic associations between the defects, instead of just their mathematical relations.

Although an overwhelming number of similarity coefficients exist, and customized coefficients can be designed for FlexTax, using one coefficient to express all the properties is not practical. To account for the properties expressed above, six different similarity coefficients were used for FlexTax. Of the six, two (Trivial and Distribution) were devised explicitly for FlexTax, one (Modified Hamming) was modified from a popular coefficient to suit FlexTax and the other three (Hamming, Jaccard-Needham and Tarantula) were adopted from popular coefficients used in previous research [30, 21].

Trivial Similarity

This similarity is relevant to FlexTax and was developed considering the specific nature of FlexTax. FlexTax recognizes subjectivity by incorporating the supervisor's discretion in the feature (attribute) generation and perspective selection. This method results in a similarity imposed on the different perspectives.

In the most trivial form, defect similarity can be expressed as a defect class's sharing of description to other defect classes. For a hierarchical taxonomy, by principle, classes in every level generalizes the classes immediately below it. Thus, any level can be considered as a group of similar classes under a perspective. In this approach, the lower a level is in the hierarchy, the stronger similarity it expresses for classes below it, and for classes under the same level. A problem with this approach is that it often does not apply to matrices or flat taxonomies.

As FlexTax assigns weight in an incremental way, any two classes sharing similar features are expected to be neighbours. Considering these facts, the trivial similarity coefficient S_T between classes c_i and c_j is defined as the compliance to two different situations. (a) if c_i and c_j are under the same perspective 1 class, and (b) if c_i and c_j are neighbours (i.e., they occur in order with one another in consideration of the weights). To avoid an undefined value problem, every class is considered as a neighbour to itself.

A way to implement such a similarity coefficient is to consider 1 point for compliance to each of the two cases described above, dividing by the maximum number of points to normalize the result within the range [0, 1]. For a taxonomy with n levels, there can be at most n-1 ancestral points for a class on the lowest layer. Additionally, there can be one more point for the neighbourhood of the classes. Considering the present scenario, the trivial similarity coefficient can be devised as,

$$S_T(c_i, c_j) = \frac{s_n(c_i, c_j) + \sum_{x=1}^{n-1} s_a(c_i, c_j)}{n} \tag{3.5}$$

Where,
S_T = Trivial Similarity Score
s_n = Similarity by neighbourhood
s_a = Similarity by ancestry.

$$s_n(c_i, c_j) = \begin{cases} 1 & \text{if } c_i \text{ and } c_j \text{ are neighbours} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

and,

$$s_a(c_i, c_j) = \begin{cases} 1 & \text{if } c_i \text{ and } c_j \text{ have the same ancestor,or if their ancestors belong to the same ancestor} \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

Table 3.4: Values for the Trivial Similarity Coefficient

		$S_T(Row, Column)$																					
#	Class	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	C1	1.00	1.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
2.	C2	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
3.	C3	0.50	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
4.	L1	0.00	0.00	0.00	1.00	1.00	0.50	0.50	0.50	0.50	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
5.	L2	0.00	0.00	0.00	1.00	1.00	1.00	0.50	0.50	0.50	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
6.	L3	0.00	0.00	0.00	0.50	1.00	1.00	1.00	0.50	0.50	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
7.	L4	0.00	0.00	0.00	0.50	0.50	1.00	1.00	1.00	0.50	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
8.	L5	0.00	0.00	0.00	0.50	0.50	0.50	1.00	1.00	1.00	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
9.	L6	0.00	0.00	0.00	0.50	0.50	0.50	0.50	1.00	1.00	1.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10.	L7	0.00	0.00	0.00	0.50	0.50	0.50	0.50	0.50	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
11.	L8	0.00	0.00	0.00	0.50	0.50	0.50	0.50	0.50	0.50	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
12.	M1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
13.	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
14.	M3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
15.	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.50	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
16.	D1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.50	0.50	0.50	0.00	0.00	
17.	D2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.50	0.50	0.00	0.00	
18.	D3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	1.00	1.00	1.00	0.50	0.00	0.00	
19.	D4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.50	1.00	1.00	1.00	0.00	0.00	
20.	D5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.50	0.50	1.00	1.00	0.00	0.00	
21.	S1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	
22.	S2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	

Note: Similarities are ordered by rows.

Table 3.4 shows the values obtained by using the trivial similarity equations over the feature set of the taxonomy generated using FlexTax.

Similarity by Defect Distribution

As the classes are defined through the same feature space, their relative spatial positioning in the same prioritized plane is under similar measures. By the definition of strong similarity provided above, a defect would only be mapped to another class should its present one be absent if the other class is a neighbour of the present one. Considering these two properties, in addition to the previous trivial coefficient, another trivial and subjective similarity measure is used for the taxonomy.

This approach for defect similarity was examined by considering the second closest defect class for every defect. As defects are assigned to the class closest to them by weight, the classes the defects were most likely to belong to in absence of their present classes make similar classes to the present one.

To assess the similarity in this way, the second closest class for every defect was considered. The similarity score by distribution, S_D, between c_i and c_j is defined as the percentage of defects under c_i that would assign to c_j in absence of c_i under the same value of T_p. For any c_i, c_i combination, the score is at 1.0. Table 3.5

presents the data on the similarity by distribution, S_D, study.

Table 3.5: Values for the Distribution Similarity Coefficient

		$S_D(Row, Column)$																						
#	Class	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	C1	1.00	0.21	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
2.	C2	0.27	1.00	0.22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
3.	C3	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
4.	L1	0.00	0.00	0.00	1.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
5.	L2	0.00	0.00	0.00	0.50	1.00	0.23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
6.	L3	0.00	0.00	0.00	0.00	0.11	1.00	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
7.	L4	0.00	0.00	0.00	0.00	0.00	0.27	1.00	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
8.	L5	0.00	0.00	0.00	0.00	0.00	0.00	0.31	1.00	0.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
9.	L6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33	1.00	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10.	L7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	1.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
11.	L8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.43	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
12.	M1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
13.	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
14.	M3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.35	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
15.	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.21	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
16.	D1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.50	0.00	0.00	0.00	0.00	0.00	
17.	D2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.34	1.00	0.70	0.00	0.00	0.00	0.00	
18.	D3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	1.00	0.17	0.00	0.00	0.00	
19.	D4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.44	1.00	0.30	0.00	0.00	
20.	D5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.22	1.00	0.00	0.00	
21.	S1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.30	
22.	S2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	1.00	

Note: Similarities are ordered by rows.

Similarity by Shared Features (Hamming Similarity)

Previous two similarities were carried out on the subjective arrangement of the defect classes. As subjectivity differs for different projects and as FlexTax uses a unique way to incorporate subjectivity, the similarity measures had to be custom designed.

To provide an objective assessment for different defect class similarities, an established method of comparison is required. As FlexTax represents the defect classes by their features (as expressed through attributes), and classifies all defect classes under the same feature space, it is possible to apply a number of feature vector comparison techniques.

Among the many comparison techniques presented in [30], Hamming Similarity (1950) [67] is perhaps the most preferred method for binary feature vector similarity assessment [30]. Although more than half a century old, Hamming Similarity was used by a number of research until recent times [164, 21, 30, 42]. However, the range for the Hamming Similarity is $[0, \infty]$. To scale it down to the preferred range of [0,1], the Normalized Hamming Similarity based on the Normalized Hamming Distance developed by Sokal and Michener (1958) [145] for biological studies was used. Hamming Similarity is noted for its straightforward feature count based comparison and applicability to binary vectors.

Hamming Similarity can be expressed using a variety of mathematical notations. In the context of FlexTax, considering the notations specified earlier, the Hamming Similarity Coefficient would be,

$$S_H(c_i, c_j) = \sum_{x=1}^{m} \frac{|\{(b_x \in B_i^c) = 1\}\ AND\ \{(b_x \in B_j^c) = 1\}|}{m} \tag{3.8}$$

Where,

$S_H(c_i, c_j)$ = Hamming Similarity between classes c_i and c_j.

B_i^c = Compliance Vector for c_i.

m = Number of Attributes.

Table 3.6: Values for the Hamming Similarity Coefficient

		$S_H(Row, Column)$																					
#	Class	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	C1	0.15	0.03	0.03	0.02	0.00	0.00	0.03	0.00	0.00	0.00	0.03	0.03	0.00	0.00	0.00	0.02	0.10	0.03	0.00	0.00	0.00	0.00	
2.	C2	0.03	0.07	0.02	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.02	0.02	0.00	0.00	0.03	0.03	0.00	0.00	0.00	0.00	0.00	0.00	
3.	C3	0.03	0.02	0.08	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.02	0.03	0.00	0.00	0.02	0.02	0.03	0.00	0.00	0.00	0.00	0.00	
4.	L1	0.02	0.00	0.00	0.10	0.02	0.00	0.08	0.00	0.00	0.02	0.02	0.02	0.00	0.00	0.00	0.00	0.03	0.07	0.00	0.00	0.00	0.00	
5.	L2	0.00	0.00	0.00	0.02	0.05	0.02	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
6.	L3	0.00	0.00	0.00	0.00	0.02	0.05	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
7.	L4	0.03	0.02	0.03	0.08	0.03	0.03	0.59	0.02	0.07	0.00	0.34	0.05	0.00	0.03	0.03	0.08	0.08	0.12	0.03	0.08	0.07	0.03	
8.	L5	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.05	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.02	0.00	0.00	
9.	L6	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.07	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10.	L7	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	
11.	L8	0.03	0.02	0.02	0.02	0.00	0.00	0.34	0.02	0.07	0.00	0.39	0.00	0.00	0.00	0.00	0.07	0.07	0.10	0.03	0.07	0.07	0.03	
12.	M1	0.03	0.02	0.03	0.02	0.00	0.00	0.05	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.02	0.00	0.02	0.02	0.00	0.00	0.00	0.00	
13.	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
14.	M3	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
15.	M4	0.00	0.03	0.02	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.07	0.02	0.02	0.00	0.00	0.00	0.00	0.00	
16.	D1	0.02	0.03	0.02	0.00	0.00	0.00	0.08	0.02	0.00	0.00	0.07	0.00	0.00	0.00	0.02	0.10	0.00	0.02	0.00	0.03	0.00	0.00	
17.	D2	0.10	0.00	0.03	0.03	0.00	0.00	0.08	0.00	0.00	0.00	0.07	0.02	0.00	0.00	0.02	0.00	0.20	0.08	0.00	0.00	0.00	0.00	
18.	D3	0.03	0.00	0.00	0.07	0.00	0.00	0.12	0.00	0.00	0.02	0.10	0.02	0.00	0.00	0.00	0.02	0.08	0.17	0.00	0.02	0.00	0.00	
19.	D4	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	
20.	D5	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.02	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.03	0.00	0.02	0.00	0.08	0.00	0.00	
21.	S1	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.03	
22.	S2	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.03	

Note: Similarities are ordered by rows.

Table 3.6 shows the values for the similarity coefficients obtained from the feature set of the taxonomy generated by FlexTax.

Similarity by Common Features (Modified Hamming Similarity)

For every defect class, only a few features from the entire set are complied with. Comparing the similarity with the entire compliance vector thus would result in very low similarity scores and would fail to capture the actual relationship, as it is apparent from the S_H coefficient in the previous section.

To avoid this problem, a modified definition by feature count is used. In this definition, the score is normalized not by the entire feature vector size, but with the specific defect's feature vector compliance. This score, S_{MH} would be,

$$S_{MH}(c_i, c_j) = \frac{\sum_{x=1}^{m} |\{(b_x \in B_i^c) = 1\}\ AND\ \{(b_x \in B_j^c) = 1\}|}{\sum_{x=1}^{m} |\{(b_x \in B_i^c) = 1\}|} \tag{3.9}$$

Where,

$S_{MH}(c_i, c_j)$ = Modified Hamming similarity coefficient between c_i and c_j

B_i^c = Compliance Vector for c_i

m = Number of Attributes.

Table 3.7: Values for the Modified Hamming Similarity Coefficient

		$S_{MH}(Row, Column)$																					
#	Class	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	C1	1.00	0.22	0.22	0.11	0.00	0.00	0.22	0.00	0.00	0.00	0.22	0.22	0.00	0.00	0.00	0.11	0.67	0.22	0.00	0.00	0.00	0.00	
2.	C2	0.50	1.00	0.25	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.25	0.25	0.00	0.00	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	
3.	C3	0.40	0.20	1.00	0.00	0.00	0.00	0.40	0.00	0.00	0.00	0.20	0.40	0.00	0.00	0.20	0.20	0.40	0.00	0.00	0.00	0.00	0.00	
4.	L1	0.17	0.00	0.00	1.00	0.17	0.00	0.83	0.00	0.00	0.17	0.17	0.17	0.00	0.00	0.00	0.00	0.33	0.67	0.00	0.00	0.00	0.00	
5.	L2	0.00	0.00	0.00	0.33	1.00	0.33	0.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
6.	L3	0.00	0.00	0.00	0.00	0.33	1.00	0.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
7.	L4	0.06	0.03	0.06	0.14	0.06	0.06	1.00	0.03	0.11	0.00	0.57	0.09	0.00	0.06	0.06	0.14	0.14	0.20	0.06	0.14	0.11	0.06	
8.	L5	0.00	0.00	0.00	0.00	0.00	0.00	0.33	1.00	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.33	0.00	0.00	
9.	L6	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10.	L7	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.00	0.00	0.00	
11.	L8	0.09	0.04	0.04	0.04	0.00	0.00	0.87	0.04	0.17	0.00	1.00	0.00	0.00	0.00	0.00	0.17	0.17	0.26	0.09	0.17	0.17	0.09	
12.	M1	0.40	0.20	0.40	0.20	0.00	0.00	0.60	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.20	0.00	0.20	0.20	0.00	0.00	0.00	0.00	
13.	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
14.	M3	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
15.	M4	0.00	0.50	0.25	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.25	0.00	0.00	1.00	0.25	0.25	0.00	0.00	0.00	0.00	0.00	
16.	D1	0.17	0.33	0.17	0.00	0.00	0.00	0.83	0.17	0.00	0.00	0.67	0.00	0.00	0.00	0.17	1.00	0.00	0.17	0.00	0.33	0.00	0.00	
17.	D2	0.50	0.00	0.17	0.17	0.00	0.00	0.42	0.00	0.00	0.00	0.33	0.08	0.00	0.00	0.08	0.00	1.00	0.42	0.00	0.00	0.00	0.00	
18.	D3	0.20	0.00	0.00	0.40	0.00	0.00	0.70	0.00	0.00	0.10	0.60	0.10	0.00	0.00	0.00	0.10	0.50	1.00	0.00	0.10	0.00	0.00	
19.	D4	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	
20.	D5	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.20	0.00	0.00	0.80	0.00	0.00	0.00	0.00	0.40	0.00	0.20	0.00	1.00	0.00	0.00	
21.	S1	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.50	
22.	S2	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	

Note: Similarities are ordered by rows.

Table 3.7 shows the result of the comparison under the Modified Hamming Similarity Coefficient.

Similarity by Jaccard-Needham Coefficient

The Jaccard-Needham Similarity Coefficient is another measure that was used in different fields of science for a long time [30, 21]. Jaccard-Needham measure was used in this study for similarity due to its treatment of the defect vectors as sets, ignoring repetitive patterns. As FlexTax's attribute set is not a repeated collection, rather a set containing only unique elements, the Jaccard-Needham coefficient is able to predict the similarities. The general form of the Jaccard-Needham similarity coefficient is,

$$S_J(c_i, c_j) = \frac{|B_i^c \cap B_j^c|}{|B_i^c \cup B_j^c|} \tag{3.10}$$

Where,

$S_J(c_i, c_j)$ = Jaccard-Needham Similarity Coefficient for classes c_i and c_j

B_i^c = Compliance vector for class c_i

Table 3.8 shows the similarity score under the Jaccard-Needham Similarity Coefficient.

Table 3.8: Values for the Jaccard-Needham Similarity Coefficient

		$S_J(Row, Column)$																					
#	Class	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	C1	1.00	0.18	0.17	0.07	0.00	0.00	0.05	0.00	0.00	0.00	0.07	0.17	0.00	0.00	0.00	0.07	0.40	0.12	0.00	0.00	0.00	0.00	
2.	C2	0.18	1.00	0.13	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.04	0.13	0.00	0.00	0.33	0.25	0.00	0.00	0.00	0.00	0.00	0.00	
3.	C3	0.17	0.13	1.00	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.04	0.25	0.00	0.00	0.13	0.10	0.13	0.00	0.00	0.00	0.00	0.00	
4.	L1	0.07	0.00	0.00	1.00	0.13	0.00	0.14	0.00	0.00	0.10	0.04	0.10	0.00	0.00	0.00	0.00	0.13	0.33	0.00	0.00	0.00	0.00	
5.	L2	0.00	0.00	0.00	0.13	1.00	0.20	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
6.	L3	0.00	0.00	0.00	0.00	0.20	1.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
7.	L4	0.05	0.03	0.05	0.14	0.06	0.06	1.00	0.03	0.11	0.00	0.53	0.08	0.00	0.06	0.05	0.14	0.12	0.18	0.06	0.14	0.11	0.06	
8.	L5	0.00	0.00	0.00	0.00	0.00	0.00	0.03	1.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.14	0.00	0.00	
9.	L6	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.00	1.00	0.00	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10.	L7	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	
11.	L8	0.07	0.04	0.04	0.04	0.00	0.00	0.53	0.04	0.17	0.00	1.00	0.00	0.00	0.00	0.00	0.16	0.13	0.22	0.09	0.17	0.17	0.09	
12.	M1	0.17	0.13	0.25	0.10	0.00	0.00	0.08	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.13	0.00	0.06	0.07	0.00	0.00	0.00	0.00	
13.	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
14.	M3	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
15.	M4	0.00	0.33	0.13	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.00	0.13	0.00	0.00	1.00	0.11	0.07	0.00	0.00	0.00	0.00	0.00	
16.	D1	0.07	0.25	0.10	0.00	0.00	0.00	0.14	0.13	0.00	0.00	0.16	0.00	0.00	0.00	0.11	1.00	0.00	0.07	0.00	0.22	0.00	0.00	
17.	D2	0.40	0.00	0.13	0.13	0.00	0.00	0.12	0.00	0.00	0.00	0.13	0.06	0.00	0.00	0.07	0.00	1.00	0.29	0.00	0.00	0.00	0.00	
18.	D3	0.12	0.00	0.00	0.33	0.00	0.00	0.18	0.00	0.00	0.07	0.22	0.07	0.00	0.00	0.00	0.07	0.29	1.00	0.00	0.07	0.00	0.00	
19.	D4	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	
20.	D5	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.14	0.00	0.00	0.17	0.00	0.00	0.00	0.00	0.22	0.00	0.07	0.00	1.00	0.00	0.00	
21.	S1	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.00	0.00	0.00	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.50	
22.	S2	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	1.00	

Note: Similarities are ordered by rows.

Similarity by the Tarantula Coefficient

Among the established coefficients considered, Tarantula is a recent addition. Tarantula is a defect localization technique that uses the customized similarity coefficient to differentiate between passed and failed execution profiles. Although not developed for a binary feature vector comparison tool Tarantula's workflow requires comparison the passed and failed profiles as binary feature vectors to determine the failure pattern.

The Tarantula Coefficient was developed to establish a semantic relationship among the passed and failed execution profiles. The technique was chosen for the current assessment due to this affiliation with semantic aspects. Tarantula Coefficient is possibly the technique to consider all forms of binary feature vector matches and mismatches most extensively. The formation of the coefficient is,

$$S_{TT} = \frac{p1}{p1 + p2} \tag{3.11}$$

$$p1 = \sum_{x=1}^{m} \frac{|(b_x \in B_i^c) = 1 \ AND \ (b_x \in B_j^c) = 1|}{|(b_x \in B_i^c) = 1 \ AND \ (b_x \in B_j^c) \neq 1|} \tag{3.12}$$

$$p2 = \sum_{x=1}^{m} \frac{|(b_x \in B_i^c) = 1 \ AND \ (b_x \in B_j^c) \neq 1|}{|(b_x \in B_i^c) \neq 1 \ AND \ (b_x \in B_j^c) \neq 1|} \tag{3.13}$$

Where,

m = Number of Attributes

b_x = One specific dimension in the compliance vector B.

B_i^c = Compliance vector for class c_i.

Table 3.9: Values for the Tarantula Similarity Coefficient

		$S_{TT}(Row, Column)$																					
#	Class	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	C1	1.00	0.80	0.76	0.52	0.00	0.00	0.16	0.00	0.00	0.00	0.31	0.76	0.00	0.00	0.00	0.52	0.89	0.58	0.00	0.00	0.00	0.00	
2.	C2	0.85	1.00	0.78	0.00	0.00	0.00	0.19	0.00	0.00	0.00	0.34	0.78	0.00	0.00	0.93	0.90	0.00	0.00	0.00	0.00	0.00	0.00	
3.	C3	0.79	0.77	1.00	0.00	0.00	0.00	0.31	0.00	0.00	0.00	0.28	0.88	0.00	0.00	0.77	0.69	0.72	0.00	0.00	0.00	0.00	0.00	
4.	L1	0.53	0.00	0.00	1.00	0.79	0.00	0.77	0.00	0.00	0.68	0.24	0.68	0.00	0.00	0.00	0.00	0.66	0.91	0.00	0.00	0.00	0.00	
5.	L2	0.00	0.00	0.00	0.82	1.00	0.90	0.58	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
6.	L3	0.00	0.00	0.00	0.00	0.90	1.00	0.58	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
7.	L4	0.25	0.29	0.40	0.60	0.53	0.53	1.00	0.35	0.64	0.00	0.68	0.50	0.00	0.63	0.45	0.60	0.39	0.55	0.63	0.64	0.64	0.63	
8.	L5	0.00	0.00	0.00	0.00	0.00	0.00	0.26	1.00	0.00	0.00	0.44	0.00	0.00	0.00	0.00	0.82	0.00	0.00	0.00	0.84	0.00	0.00	
9.	L6	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10.	L7	0.00	0.00	0.00	0.69	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.55	0.00	0.00	0.00	0.00	
11.	L8	0.35	0.38	0.33	0.29	0.00	0.00	0.82	0.46	0.74	0.00	1.00	0.00	0.00	0.00	0.00	0.65	0.45	0.63	0.73	0.69	0.74	0.73	
12.	M1	0.79	0.77	0.88	0.69	0.00	0.00	0.51	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.77	0.00	0.49	0.55	0.00	0.00	0.00	0.00	
13.	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
14.	M3	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
15.	M4	0.00	0.93	0.78	0.00	0.00	0.00	0.41	0.00	0.00	0.00	0.00	0.78	0.00	0.00	1.00	0.75	0.57	0.00	0.00	0.00	0.00	0.00	
16.	D1	0.53	0.87	0.68	0.00	0.00	0.00	0.77	0.79	0.00	0.00	0.76	0.00	0.00	0.00	0.73	1.00	0.00	0.49	0.00	0.84	0.00	0.00	
17.	D2	0.85	0.00	0.68	0.64	0.00	0.00	0.33	0.00	0.00	0.00	0.44	0.50	0.00	0.00	0.56	0.00	1.00	0.78	0.00	0.00	0.00	0.00	
18.	D3	0.58	0.00	0.00	0.85	0.00	0.00	0.62	0.00	0.00	0.55	0.70	0.55	0.00	0.00	0.00	0.50	0.80	1.00	0.00	0.55	0.00	0.00	
19.	D4	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	
20.	D5	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.82	0.00	0.00	0.86	0.00	0.00	0.00	0.00	0.85	0.00	0.55	0.00	1.00	0.00	0.00	
21.	S1	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.97	
22.	S2	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	

Note: Similarities are ordered by rows.

Table 3.9 shows the similarity score under the Tarantula Similarity Coefficient.

3.9.2 Similarity in Defect Classes

Figures 3.6 - 3.10 show the defect class similarity over the coefficients presented in earlier sections. The indicators ST (S_T), SD (S_D), SH (S_H), SMH (S_{MH}), SJ (S_J) and STT (S_{TT}) denote the Trivial, Distribution, Hamming, Modified Hamming, Jaccard-Needham and Tarantula similarity coefficients.

Subjective indices like S_T and S_D do not rely on the feature space, rather incorporates the subjective notion used by the taxonomy developer, while S_{MH}, S_J and S_{TT} use the objective feature space and / or semantic association. As can be seen from the graph, the S_{MH}, S_J and S_{TT} are more predictive (display more variance) than the S_T and S_D. S_H's performance should be with those of S_{MH}, S_J and S_{TT}, but it is not noticeable due to the low coefficient values typical for Hamming Coefficients (which is the reason S_{MH} was developed for this study).

The defect class C1 is predicted to have strong association with M1 and D2, with a weak association with L4, L8, D1 and D3, in addition to being associated with C2 and C3. C2 is associated with C3 by most coefficients, and with M1, M4 and D1 by some with a weak association with L4. C3's association is almost the same as C2.

L1 is connected with a number of classes spanning C, L, M and D groups, as it is the underlying reason

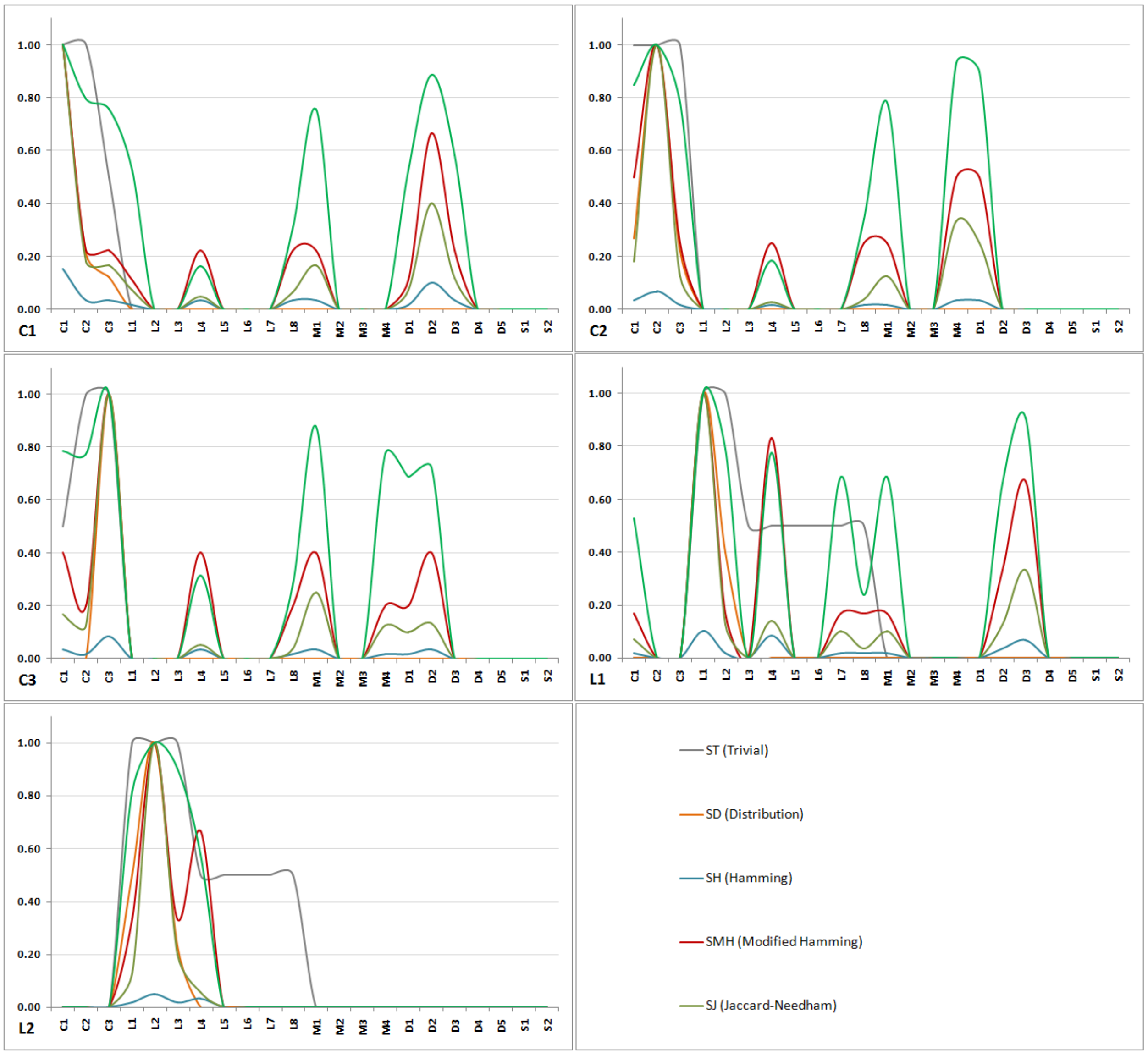


Figure 3.6: Defect Class Similarity Over Different Coefficients

for most defects. L2 and L3 are reported to be rather localized over the other logical classes. L4, on the other hand, can be the underlying cause for most of the defects and is associated with virtually every other class. Same observation holds for L8, with the exception of M2 and M3. L5 can result in D1 and D5, and thus is strongly associated with them. For L6, the reports vary as the "Improper Exception Handling" can be resulted from a variety of other causes. L7 is mostly associated with L8, and L1, as it is a somewhat special case of L1.

For the memory defects, M1 is associated with a variety of other classes, as an invalid memory reference can be from a number of issues. M2 is not associated with any defect but itself, while M3 can be a direct result if L4, and thus is associated with it. M4, again, is possibly one of the most populous defect class in existence and can be associated with a number of other classes.

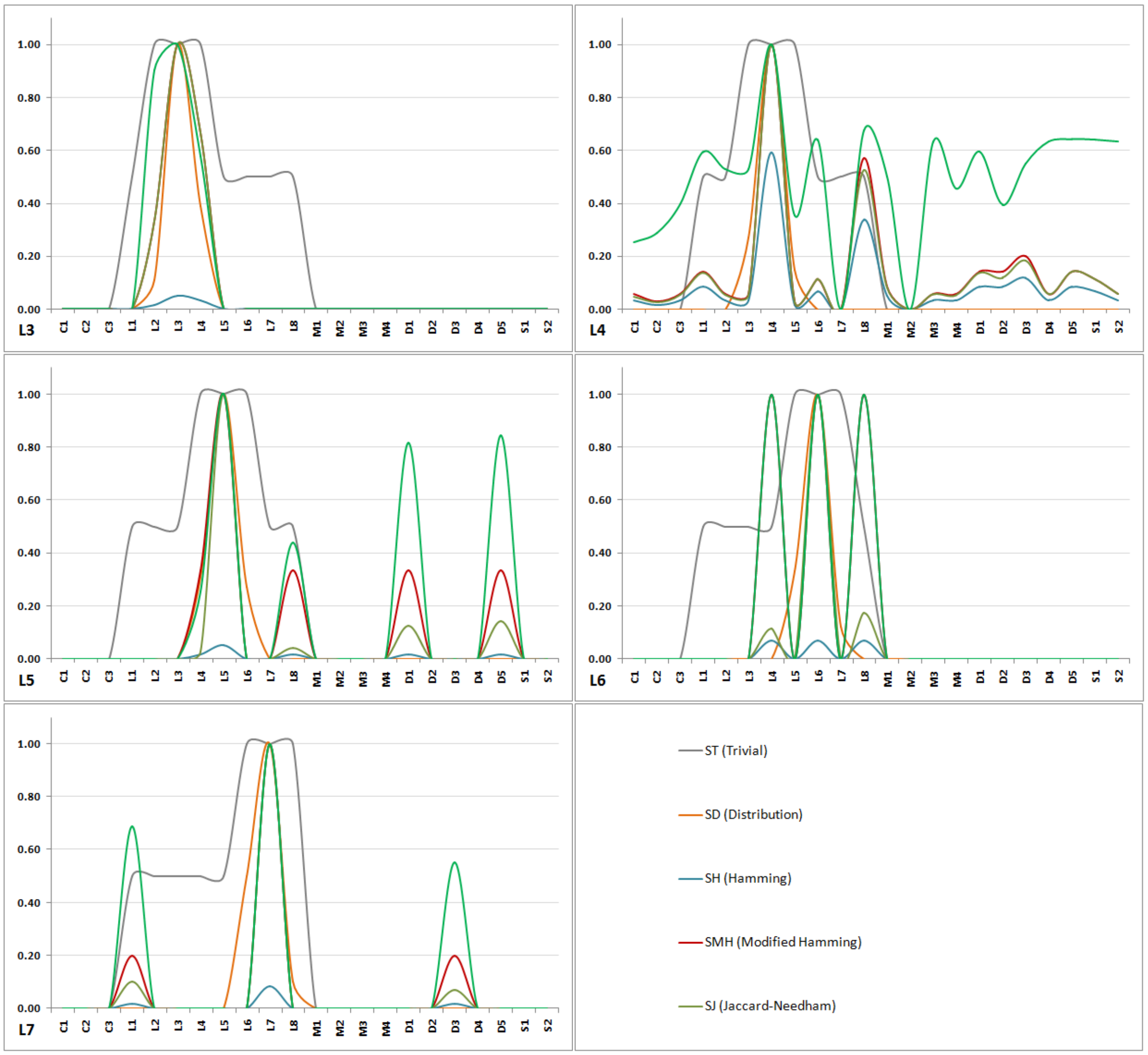


Figure 3.7: Defect Class Similarity Over Different Coefficients (Continued)

Data related defects are usually associated with C, L and M groups, as they can be the outcome of defects belonging to the C, L and M groups, as it is reflected by the similarity coefficients.

S1 and S2 are related to each other, with weak relation to the L4 and L8 defects.

One interesting observation can be made from the graphs. The S_T and S_D coefficients are modelled after the same subjective interests that were used to develop the two perspectives of the taxonomy. It is expected for S_T and S_D to correspond to the classification (that is, grouping C1-3, L1-8, M1-4 and D1-5 and S1-S2 in four separate similarity groups). The other four classifiers rely on the underlying feature set and do not contain any relation to the subjective discretions. Interestingly, all four of the coefficients, S_H, S_{MH}, S_J and S_{TT} showed similarity in the same groups, supporting the incorporation of proper similarity by subjective means in the development phase.

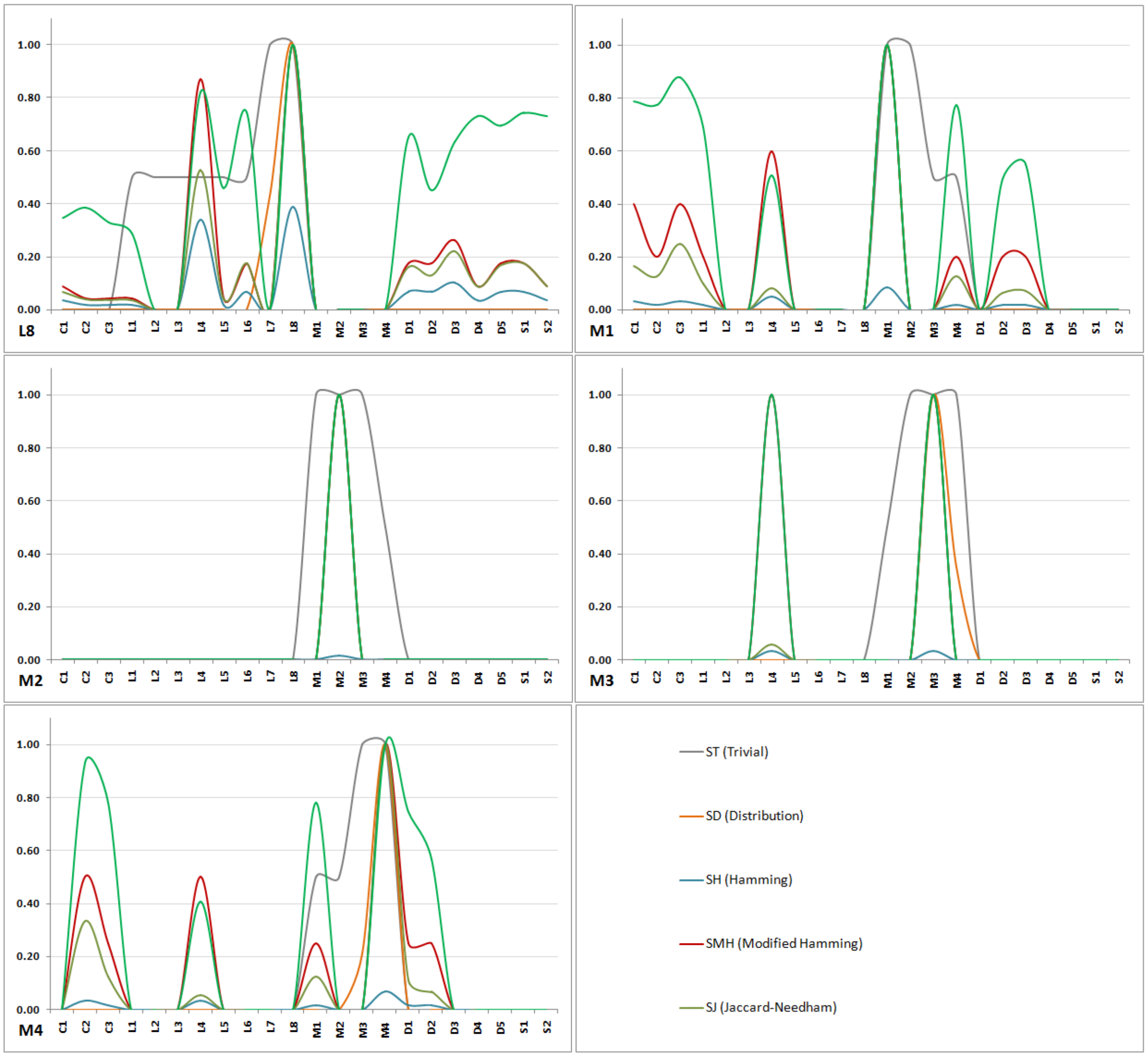


Figure 3.8: Defect Class Similarity Over Different Coefficients (Continued)

3.10 Answering the Research Question

This section answers one of the five research questions presented in Chapter 1.

3.10.1 RQ5: Defect Similarity Criteria and Procedures

The last among the five research question presented in Chapter 1 was stated as, “What are the criteria and procedure to determine similarity in defect classes?”. To answer this question, the terms ‘defect class’ and ‘similarity’ need to be defined. This chapter described the definition of defect classes, and then detailed a framework, FlexTax, used for generating defect taxonomies. FlexTax’s representation of defect classes was based on defect features and feature similarities - facilitating the similarity comparison.

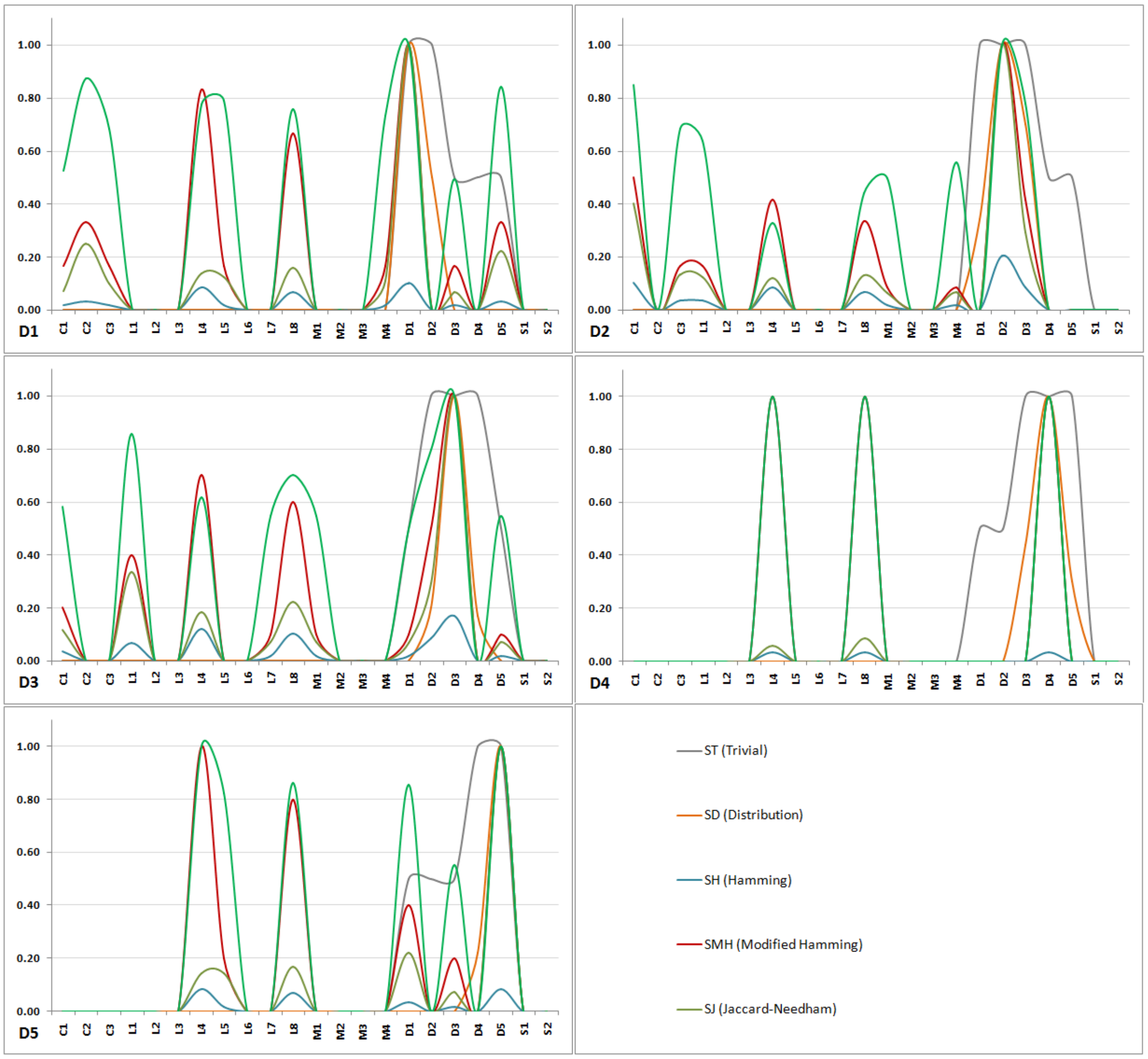


Figure 3.9: Defect Class Similarity Over Different Coefficients (Continued)

Although the similarity study conducted in this research was a primary step to the the task, it nevertheless was able to point out the similarity and dissimilarity in different classes of defects. The similarity study showed than it is possible to utilize different feature representation and comparison metrics to obtain a measure of similarity among defect classes, and that this similarity can be expressed quantitatively indicating various degrees.

In summary, the information presented in this chapter outlined one possible way of finding out defect similarity. According to the studies reported in this chapter, defect class similarity can be expressed by representing defect classes by their feature set then by applying established feature comparison metrics to compute the similarity.

This particular topic is recognized by this research as a open research question that requires further

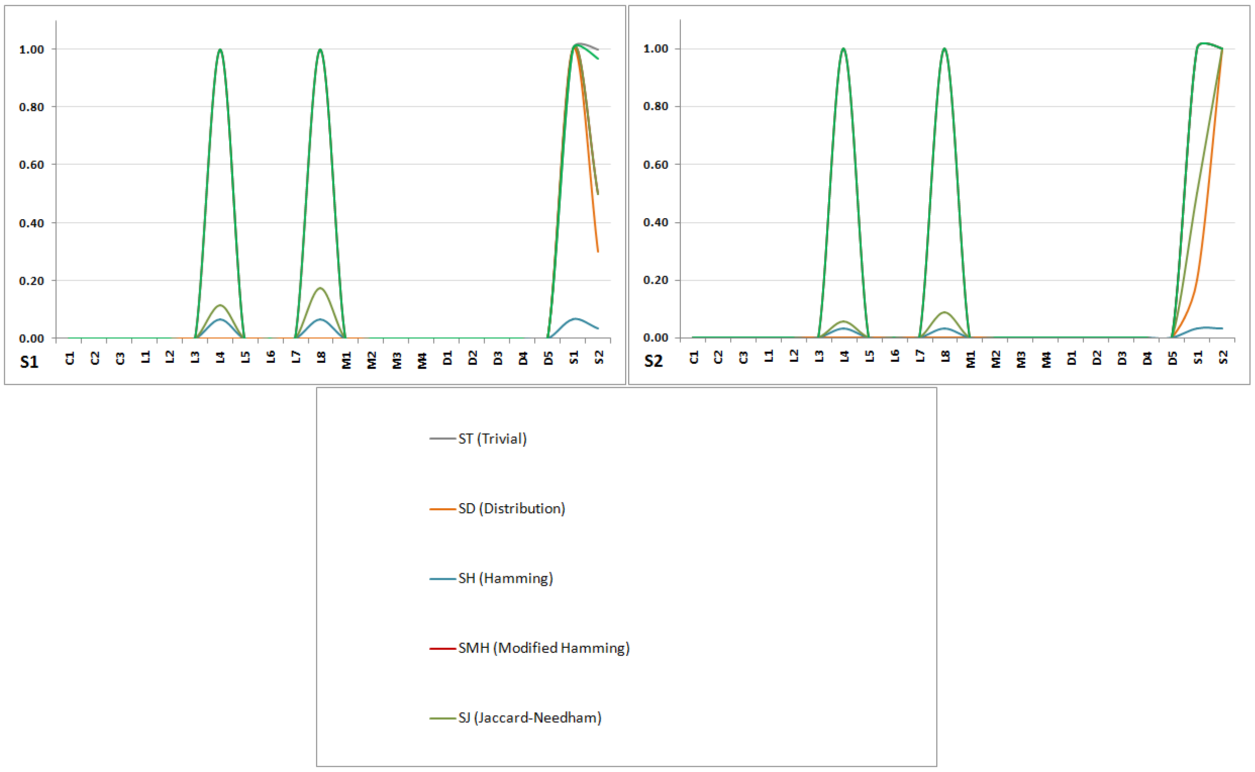


Figure 3.10: Defect Class Similarity Over Different Coefficients (Continued)

investigation. Although a procedure was set up by this research that answers the research question, it still requires further analysis.

3.11 Summary

This chapter described the state-of-the-art in defect taxonomies, a proposal of a novel taxonomy generation and defect mapping techniques, experience of using the technique to develop a taxonomy over large scale real world dataset, evaluation of the taxonomy over multiple strategies, and a study to describe defect similarity. The chapter started by analyzing notable existing defect taxonomies, selected under defined criteria and presents a qualitative comparison of the taxonomies - from which the major weaknesses in existing taxonomies were deducted. The chapter then proceeded on describing a semi-automatic framework named FlexTax that was developed to address the specific weaknesses of taxonomies and which generates taxonomies while mapping defects in them. Next, the chapter described a case study conducted to validate FlexTax, and described the taxonomy generated from the case study. Finally, the chapter described defect similarity measures and evaluated the taxonomy against the defect similarity measures. The taxonomy and defect similarity measures described in this chapter are later used in Chapter 4 to compare and assess existing techniques, in Chapter 5 to design the new technique of defect detection, and in Chapter 6 to provide a common and consistent framework for comparison and evaluation.

Chapter 4
Existing Tools and Techniques

This chapter provides an overview of the state-of-the-art in defect detection through code analysis using the defect taxonomy established in Chapter 3. The chapter begins by describing different solution strategies used in defect detection (Section 4.1). Later sections describe the strategies, and techniques under the strategies. As the focus of the current research is on Symbolic Analysis, this chapter describes Static and Symbolic Analysis techniques in greater depth. Other techniques, and tools belonging to such techniques, are mentioned with brief descriptions. The description is ordered with techniques, mentioning a tool under all relevant techniques. In case a tool employs a number of techniques, its description is split over the sections covering different techniques, without repeating the same information. General information like the tool's structure or capabilities are specified in the first mention of the tool in usual reading order. SRTA, the technique developed in the current research, is described later in Chapter 5. But considering its relevance, and to compare it with existing tools, elements of SRTA's description have been incorporated in this chapter.

4.1 Solution Strategies

Solution Strategies for defect detection fall into three broad categories - Manual Inspection, Static Analysis and Dynamic Analysis [96]. Static and Dynamic Analyses have been adopted and combined into a number of other strategies like Symbolic Analysis, Statistical Debugging or Verification.

Figure 4.1 summarizes the broad categorization of techniques that are discussed in this chapter. In the Figure, solid arrows denote a 'uses' relation from the user to the used. Dotted arrows denote a 'is-a' relation from the derived to the source. Subsequent sections describe the techniques (and include the tools) for the strategies of interest.

4.1.1 Manual Inspection

Manual inspection, or code review, is the process where the software source code is inspected by human inspectors, with or without the help of automated tools, to check their compliance to preset standards or common practices. The actual process often involves checking the code against a defined checklist [1]. Despite its strictly human-centric nature and high requirement for human labour [1], manual inspection is still a major activity in software quality assurance, especially in modern agile development methodologies.

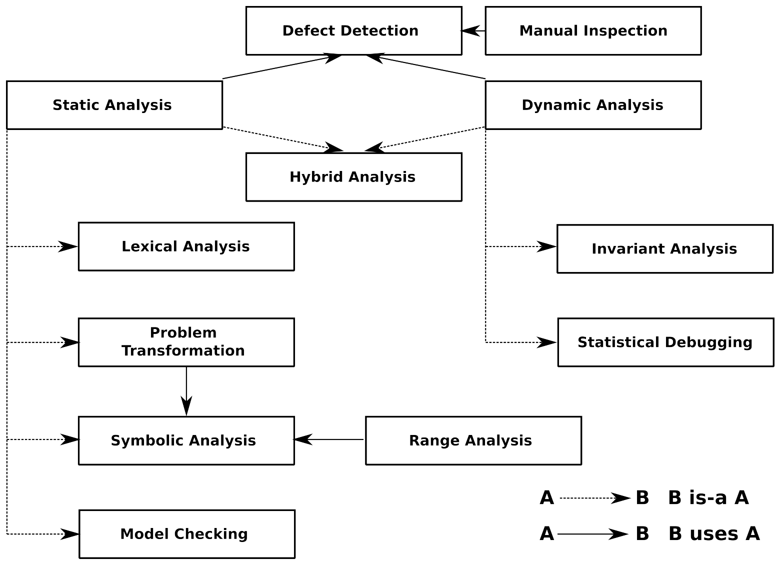


Figure 4.1: Strategies Used in Detecting Defects

Research concerning manual inspection covers three main directions. The first direction concerns assessing the effectiveness of the inspection process and measuring the impacts of different factors on the inspection process. A research by Albaryak et al. [1] measured the impact of code styles over the functional defects found by review, and concluded that the indentation and style have a significant negative impact on the detection process, although identifier naming, contrary to popular belief, was not reported to have any impact. Prause et al. measured social impacts of continuous review processes [131, 130], and concluded that collaborating development aids development quality.

Second of the three directions involves developing different automated tools to aid in the development. Efforts include a social development platform by Prause et al. [131], a lightweight visualization technique called NOSEPRINTS by Parnin et al. [125] to inspect code smells[1], and another code inspector named ICICLE, developed by Brothers et al. [18] as one of the first attempts to groupware based code inspection.

Last of the three directions directs towards developing practical code review techniques to improve its effectiveness. A series of recommendations for effective code reviews have been made by Kelly and Shepard [78], as determined by three “controlled experiments”. The recommendations outlined the best practices and effective methodologies for Manual Inspection processes. A recent publication by Bacchelli and Bird [9] investigated the practice using modern collaborative tools.

This research does not assume a direct position in the debate on the effectiveness of manual inspections, rather focuses on improving the state-of-the-art of automated defect detection which may or may not be used by manual inspection processes as supportive aids.

[1]A symptom in the source code that does not create a problem by itself, but can indicate the presence of a problem. An example is repeated code that inidicates maintenance problems.

4.1.2 Static Analysis

Static analysis analyzes the source code only, without any runtime information. This particular approach provides a number of advantages over other techniques. First, static analysis can probe any execution path in the source code, including the ones not reachable at runtime. This particular advantage enables Static Analysis to detect defects that are not yet encountered, but might surface in future once the execution path is taken. Second, as it only requires the source code, static analysis can examine software before release, or even before the software is complete enough to result in a successful build - providing defect detection in the early stages of development. In specific problems, like Buffer Overflow Detection from large legacy codebase written in unsafe languages like C and C++, researchers often consider static analysis as the only feasible approach for proper detection [168].

The most compelling advantage for static analysis is the capability of enumerating and exploring all execution paths, regardless of their invocation at a particular execution and irrespective of their possibility of being exploited ever after deployment. This property enables static analysis to be able to detect defects that might never be exposed at runtime and therefore shall be beyond the scope for Dynamic Analysis. Ironically, the capability of exploring all paths often poses itself to be the biggest drawback for static analysis in the form of so-called "path explosion", that is, the extremely large number of paths the execution has a possibility to proceed on [91, 23]. Path explosion creates a challenge for static analysis tools to perform their analyses in reasonable time and often lowers their accuracy - usually exhibited through a high false positive rate. It was reported that code complexity has a reciprocal relation with the accuracy of the static analysis result [160] and the same hypothesis was supported by an earlier experiment [168]. Often the false alarm rates for static detection tools are either difficult to assess, or are simply ignored, "possibly due to the irritation resulted from seeing too many false alarms" [168]. Sometimes program slicing is used as a countermeasure to the path explosion problem [91], despite the fact that it does not always provide a good result. Some specialized and tricky defects that depend on execution order, like concurrency bugs, cannot be detected with satisfactory accuracy by the application of pure static analysis [79].

The ability of explore all execution paths also provides the way for finding paths relevant to a specific input set - an idea exploited by the middle phase of the three phase DSD Crasher [39]. Exploiting this idea reduced the number of execution paths, although is not applicable without the aid of dynamic analysis.

A number of tools have been developed to utilize different flavours of static analysis, having utilized techniques as diverse as symbolic analysis [163, 158], abstract interpretation [129] and inter-procedural analysis [129, 163, 158]. Often the conversion of the actual codes into a separate problem domain is applied to counterbalance various problems with static analysis [163].

Static analysis was exploited in the ARray CHEckeR (ARCHER) by Xie et al. [163]. The tool is reported to evaluate array access, pointer dereference and function calls against a derived set of constraints to find violations that might result into an error or an exploitable vulnerability [163]. ARCHER does not require code annotations, works on C and C++ and is reported to handle multi-million lines of code [163], although the accuracy was in question in large scale experiments [168] and in detecting buffer overflow defects.

SAFE [57] is a tool developed for Java codes that uses both structural checks and inter-procedural flow sensitive dataflow solving techniques to find defects. In its first phase, the tool tries to detect defect signatures by constructing an XML model [57] and then proceeds on by using a type-state checking to detect trivial defects. The tool is distributed as an eclipse plug-in and is also able to function as an ANT-task or a stand alone command line tool [57]. The tool is reported to have limitations in handling large programs [57].

Lexical Analysis

Lexical analysis, as a static analysis, analyzes software properties through the set of tokens generated from the source files. This form of analysis can find defects that do not require any inference, or require very limited inference.

Evans et al. [49] developed LCLint, a tool that uses purely static analysis to detect a number of defects including violation of abstraction boundaries, data hiding, memory leaks, reference to unallocated memories and null dereferences [49, 48, 50]. The tool works on C and C++ and relies heavily on code annotation that puts its applicability to existing codebases in question [163].

Evans et al. [50], in a later work, used what they term as "lightweight static analysis" to find buffer overrun vulnerabilities from source code with SPLINT, an updated version of LCLint [49]. This tool, like its predecessor, requires code annotations and detects problems by matching code with annotations [50], although it is able to work to a degree where no annotation is provided [168]. SPLINT works on C code only and is known to detect stack and heap based buffer overflow defects - using some trivial heuristics as well as complex program value analysis to detect the likely candidates to the problem. In the words of the developers of the tool, the effort requirement for code annotation is significant [50].

Holzmann [68] developed UNO, the tool to detect Undefined outcome, Null pointer dereference and Overflows. The tool is actually a model verifier that works on C code only. The tool provides intra- and limited inter-procedural analysis under the names of "Local" and "Global" analyses. Although lightweight (by requirement) and easily deployable, UNO's accuracy was put to question by multiple experiments [29, 168].

Due to the limitation on static analysis to ensure accuracy and efficiency at the same time, approaches were proposed to balance between the two. Viega et al. [154] employed static analysis in detecting race conditions, buffer violations and other security vulnerabilities through a tool named ITS4. ITS4 works by breaking the non-preprocessed source code into a set of tokens and then analyzing them against hand-coded token sequences [154]. The tool rely entirely on code annotation to be able to process likely vulnerabilities.

4.1.3 Problem Domain Transformation

Due to the nature of the large number of program paths to analyze, often static analysis is adapted into transforming the problem involving source code to another representation specific for the type of defect the analysis intends to detect.

In 2000, Wagner et al. [158] reported a prototype that used static analysis to formulate an integer constraint problem from the source code. The prototype was intended to detect buffer overflow problems, and modelled the test subjects as a pair of integers to verify against each other [158]. A violation on the constraints would result into a potential buffer overflow identification. The prototype is scalable to systems of varying sizes, but due to the scalability-precision trade-off, the precision is sacrificed [158]. It is able to detect a subset of off-by-one errors but does not cover the entire set. A disadvantage of the prototype is its incapability of handing multiple redirection of pointers, array of pointers, function pointers and unions [158], along with its inaccurate flow-insensitive analysis [158]. The authors also proposed an algorithm to solve the integer constraint system in an efficient way using a Directed Acyclic Graph (DAG). However, the approach is reported to have suffered from very high false positive rates [168], an issue that was prevalent on a later work by Xie et al. that relied on it [163].

Ganapathy et al. [56] presented a tool generated with the combination of several other tools that uses a linear constraint generation and taint analysis for the detection of buffer overflow defects in flow and context insensitive manners [56]. Their approach uses CodeSurfer to find points-to information for the code and associates a four variable set to determine the minimum and maximum size of the allocated and used memory for a buffer [56]. The approach uses a "taint analysis" to refine the constraint set by removing the invalid variables from it. The tool used two solver modules in solving the constraints using linear programming to find the best estimates for their values. The tool is reported to be able to handle an arbitrary level of dereferencing but suffers considerably from both high false positives and high false negatives [56].

Using a finite state machine (FSM) to analyze dereferencing has been proposed by Chen et al. [24] that detects the memory dereferencing and double deallocation, and provides assumptions on the undecidable points-to analysis in representing the pointer states through an FSM. The technique detects null pointer and wild pointer errors along with a partial coverage with the double deallocation detection [24]. The technique suffers from context insensitivity and the lack of support for function pointers [24].

4.1.4 Hybrid Analysis

Hybrid analysis tries to address the problems of static and dynamic analyses by combining them into integrated techniques to use on code. Typically the application involves selecting the paths that the program is more likely to trace for static analysis, based on the data collected by dynamic analysis. A point to note is that, pure dynamic analysis do not have a way of probing into source code, unless proper instrumentation is inserted in code to generate a precise enough report. As this study focuses on source code defects, we did not provide detailed description of the pure dynamic analysis tools, although we have mentioned a few.

Often Static and Dynamic Analyses are used to refine, restrict or verify one another's outcome. This technique was utilized in Check and Crash [38], DSD Crasher [36], and HeapMD [26]. In the first two, static analysis was used to refine the results of a dynamic analysis for better performance, and on the latter, dynamic analysis results were applied on static analysis to pinpoint the problem.

Invariant Analysis

DSD Crasher [39] is a hybrid tool that combines static and dynamic analysis techniques to detect a variety of defects in source code. On its first phase, the tool detects dynamic invariants from an existing test suite using Daikon [47], thus being a subject to Daikon's requirement of a large set of test data to detect the invariants [17]. Attempts were made to overcome this by using a verification through ESC/Java [51], a tool developed earlier by two of the authors of DSD Crasher [38]. The tool uses the Daikon generated invariants in two ways, as an assumption of a method's formal parameters inside its body, and as a requirement for the actual parameters that the method is called with at its calling point [36].

Dimitrov and Zhou [45] proposed a framework that is claimed to be applicable on both error protection and defects detection, based on dynamic detection and enforcement of instruction level invariants. The approach uses a table structure named LVDV (Limited Variance of Data Values) that keeps the invariant information as collected from a successful execution of the system under test. The tool then checks the test run against the invariants to find a potential defect location.

4.1.5 Statistical Debugging

Statistical debugging is a form of dynamic analysis for identification of the cause of failures, i.e., a defect, by application of statistical models on program profiles. Almost always dependent on instrumentation, a program is required to generate a *profile* that can be monitored and analyzed statistically to locate the point of failure. However, the technique requires multiple program runs, refining the profile incrementally but never being able to provide complete information in any single run. Choosing proper instrumentation predicates (i.e. (often) boolean value generators) remain as a key problem in statistical debugging as the effectiveness of the technique depends largely on it [74]. Often established techniques like machine learning [74] are used for the selection of the predicates that accurately model the defect. The predicates are usually boolean value generators, but it was argued [4] that complex value predicates serve a better purpose. However, the approach is often criticized for introducing large execution overheads [27], and being a poor indicator of the actual defect predictors [5, 105]. Recent studies claim that it is impossible to identify complex defects automatically [132] and that, experts are likely to increase their efficiency using the techniques, when the defect is simple in nature [126].

Statistical debugging has made its way into a number of defect detection efforts. Liblit et al. [105] proposed a statistical debugging method by instrumenting predicates at particular program execution points and using a bit vector to summarize the results from all predicates after each program run. The study focuses on removing logically redundant predicates. The remaining predicates are then used in localizing defects in source code blocks. The study claims to be extended to all kinds of source code defects, provided proper predicates are chosen. The approach was more elaborated in a follow up work on adaptive bug isolation [5] that searches on the control dependence graph of the program to locate defects in the system. The latter approach is reported to significantly reduce the average performance overhead to an order of 1% as compared to the 87% of the realistic sampling based instrumentation [5].

Jiang and Su [74] proposed a path-sensitive and context-aware statistical debugging methodology with the use of Random Forests and Support Vector Machines (SVM). Their approach is to assign a classification score or weight to each predicate to measure its likeliness in predicting a defect [74]. A linear SVM classifier is employed to perform the task, later random forests are used to permute data values to each classifier. A k-means clustering algorithm is used to discover the correlation among predicates and then the relevant predicates are used to execute a branch prediction mechanism that constructs the faulty control flow directions. The control flow path prediction is a greedy algorithm traversing every possible control flow path and isolating those relating to the most weighted predicates.

Machine learning approaches were also employed through feature selection techniques by Roychowdhury and Khurshid [138]. Their approach uses well-known feature selection algorithms like RELIEF and its derivatives.

Although predicates are the most used candidates for statistical debugging, a tool named HOLMES was developed by Chilimbi et al. [27] to use path profiles instead of predicate profiles as the instrument for statistical debugging. The tool relies on the assumption that "only a small portion of the code is related to a given bug" [27] and reduces the space and execution overheads by inclining towards a defect-centered approach. The tool allows a program to run without any profiling until it encounters a failure, thus ruling defect-free programs as "free to run" without any interruption or overhead. The tool combines failure profiles and static analysis to construct a set of "likely defect containing portions" that are later profiled heavily to test against the defect. After a sufficient number of profiles are collected, statistical models are used to isolate the paths that strongly predicts a defect. The tool uses an importance score to paths for the assessment of their being predictors of the defect y a combination of three statistical metrics, *sensitivity*, *context* and *increase*. Among these, *sensitivity* was described as the logarithmic ratio of the failed executions that invoke a given path to the total failed executions, context is the ratio of the failed executions to total executions and increase is the increment of the ratio of the failed executions that invoke a given path with the total executions involving that given path over the context.

SOBER, a statistical defect isolator by Liu et al. [109], used profiles not only from the failure runs, but also from successful runs to predict a defect's location. The tool reports a defect probability only if the failure profiles differ *significantly* from the successful execution profiles.

One major direction in statistical debugging is improving the effectiveness and efficiency of the process itself. Zheng et al. [165] proposed an iterative collective voting scheme, influenced by the bi-clustering algorithms, to judge the suitability of the predicates as being bug predictors. Jiang and Su [75] apart from their original work on the path-sensitive and context aware methodology exploiting random forests and SVM [74], used the profiles as a program simplification tool to reduce the complexity of analysis. In a separate research, Liu and Han [108] proposed a new proximity metric named R-proximity to effectively group multiple related profile information that lead to the same defect.

Due to their completely non-deterministic nature, concurrency bugs remain one of the prime targets for statistical debugging. In a recent work, Lucia et al. [113] proposed *Recon*, a graph-based partially context-aware representation of inter-thread communications to reconstruct executions of concurrent programs for detecting defects that might have resulted into a failure or abnormal behaviour. According to the claim of the paper, the tool does not only detect defects, but also attempts to provide insight on why the defect occurred [113]. The tool is reported to detect single and multi-variable concurrency defects [113]. In an earlier but recent study, Jin et al. [77] presented the Cooperative Crug Isolation framework (CCI), that isolates "crugs" (concurrency bugs) using a low overhead framework. The issue of parallel programs was also addressed by Zhou, Kulkarni and Bagchi [166] through their tool Vrisha that detects defects in large scale software by using a combination of small-scale models of defect free behaviour.

4.1.6 Symbolic Analysis

Traditionally considered as a static analysis technique, symbolic analysis augments the capabilities of pure static analysis techniques by providing symbolic execution data. As the exact dataflow cannot be determined without the aid of dynamic analysis, symbolic analysis attempts to provide a similar data by approximating information that often involves constraint solving or linear programming. Symbolic analysis converts the problem into a set of symbolic values or constraints that can be resolved, or at least, guessed to find the possible outcome. Often the failure to resolve a constraint or to find an anomaly in terms of the symbolic values and a predefined rule lead to the conclusion of finding an error. Symbolic Analysis can be thought of as a special case of the Problem Domain Transformation technique mentioned in Section 4.1.3.

In the art of defect detection, probably the use of exclusive Symbolic Analysis for detecting defects started with the technique by Wagner et al. [158]. Their symbolic domain was composed of integer values derived from memory bounds, which employed a simplified constraint solver to approximate the values used in the symbolic arithmetic. Later, in ARCHER, Xie et al. [163] used a symbolic analysis technique to bind the values to variables and memory sizes. The values are represented in symbolic constraints that determine whether or not the statement qualifies for a defect [163]. The constraints are then resolved to find the actual threats. They developed a specialized solver that approximates the values from different conditions.

Li et al. [103] presented an algorithm on effective symbolic analysis using simple rules to detect buffer overflow vulnerabilities in code. The algorithm focused on data and control dependencies and instead of checking every execution path, it worked on linearly related control dependencies [103].

Symbolic analysis is exploited in the generalized technique of Structural Abstraction and Refinement (SAR) that has been employed by a number of defect detection tools [20, 144, 149]. Among them, Sinha [144] proposed a generalized Structural Abstraction Technique that analyzes program "regions" corresponding to modular constructs.

FindBugs, [69, 71, 70] a project that started with an experiment and incrementally developed into a complete product, is cited for being one of the most comprehensive defect detectors [8, 7]. The tool works for Java only, and uses symbolic analysis with the notion of extending a simple analysis technique and / or model in detecting multiple defects. Findbugs is cited to detect concurrency bugs, but actually the detection is limited in 'suspecting' a few language specific constructs provided in Java [79].

In the most recent approach, Le and Soffa [97] devised a tool called Marple that relies heavily on Symbolic Analysis. Marple is effective on four types of defects and is path-sensitive, context-sensitive and inter-procedural.

Grammatech's CodeSonar [63] is one of the most cited industry-developed tool that detects a wide variety of code defects through a symbolic execution engine. Coverity's SAVE [35] and Klocwork's Insight[81] are the two other similar tools. Although details of these tools are not public, they are claimed to have detected "hundreds of types" of defects through symbolic models.

SRTA, the tool developed in this thesis, employs purely Symbolic Analysis as its working procedure. Similar to Wagner et al. [158], ARCHER [163], and Li et al.[103], SRTA uses constraint formation and solving techniques, but differs with all three in its novel application of a 3-tuple of constraints formed as ranges providing greater flexibility, and in its capability to detect dissimilar classes of defects. Marple [96] and Findbugs [71] make the propositions SRTA builds on, and SRTA has similarity with them in detecting multiple defects and in using symbolic analysis, but differs with both in its incorporation of multiple-phases of an entity's life cycle and complete path envelopment that, collectively, increase SRTA's capabilities.

4.1.7 Context Sensitive Analysis

In detecting defects, context sensitivity usually means the analysis relative to a block of code's calling context. In extended definitions, it might include the set of all active methods in the stack [16]. It is argued that context sensitivity helps in reducing false positive by refining the results [163, 96]

ARCHER [163] uses a context sensitive analysis by using a call graph of the functions to keep track of the calling contexts. The approach utilizes a Depth First Search in transforming the control flow graph into a set of solver states, checking block by block while selecting a block's successors in random order. However, the traversal has a time constraint associated, default to five seconds [163] that breaks the analysis once exceeded.

Although their approach was not strictly context-sensitive, Ganapathy et al. [56] relied on constraint-inlining and inter-procedural dataflow analysis to incorporate a degree of context sensitivity.

The approach by Jiang and Su [74] provided an algorithm that used a context-aware statistical model for debugging using machine learning algorithms. Depending on a number of heuristics, they combined several machine learning algorithms to construct context aware processing models.

Apart from the tools and techniques that are context sensitive themselves, frameworks are proposed to add context sensitivity to those that do not already have them. Breadcrumbs [16] is a framework proposed by Bond et al. that extends dynamic defect detection tools by adding context sensitivity.

4.1.8 Path Sensitive Analysis

Path sensitive analyses, in addition to the other methodologies, consider the exact execution path that was used to reach a statement in assessing the statement's being a defect. It is argued that path sensitive analysis cannot be substituted for a scalable and accurate technique [95, 96, 97]

Marple by Le and Soffa [97] provides specific treatment for a path sensitive analysis in its detection process. The tool is reported to eliminate a number of false positives by path-sensitive analysis [97].

ARCHER [163] uses what the authors call a "traversal module" to check for infeasible paths in a program's execution space. The module checks the feasibility of paths using binary constraints and avoids the infeasible paths to improve efficiency. However, the authors claimed the accuracy to be "considerable" for their test systems, although not completely sound [163].

Often static analysis adopts one of the two approaches in handling paths. The first approach, detecting and excluding infeasible paths (e.g., [121, 167, 97]) provides scalable results, but increases the risk of false negatives in case a faulty path is excluded from the analysis. The second approach, creating summaries at specific program points instead of excluding paths (e.g., [143]), is theoretically able to provide better false negative performance, but risks the loss of information in creating summaries. Moreover, it is argued by Wei Le that summaries are not able to provide a complete analysis [97] which is in disagreement to a degree to the findings of other researchers [15, 14].

SRTA is not path-sensitive in the traditional way, rather SRTA incorporates path-sensitivity through a Hybrid Data and Control Flow Graph which is used in the processing, but unlike most other tools described in this section, do not rely on paths for the entire duration of the analysis, providing less restrictions on the interpretation of symbolic relations. In handling paths SRTA uses the approach to summarize the path information, to a degree where sufficient information is retained in summary to find the exact path taken.

4.1.9 Model Checking and Enforcement

Model Checking, or formal verification, involves the checking of the subject system against a defined set of "properties" under strict formal constraints. Although accurate and efficient, model checking requires a deep understanding of the system to be able to provide effective analysis results.

Trainin et al. [150] argued that injecting small models of conditions on concurrent program can expose concurrency defects. A model of conditions is defined as a function that takes a program as an input and generates a set of suspected conditions on program interleaving as output [150]. The approach was tested for the improper serialization defect by using two models of conditions in relatively small programs. Experimenting on their custom designed small programs with intentional concurrency bugs, the authors reported an accuracy improvement of a factor by 7 and 73, respectively [150]. An implication of this study is the proposition that having a low false positive rate is not an absolute prerequisite for a combination of tools.

Frama-C [41] is a static analysis framework that provides model checking capabilities. The framework relies on user supplied specifications to evaluate the test system (written in the C Language) against the specifications.

4.1.10 Other Techniques

A number of combination of techniques have been developed for the detection of defects that do not fall in earlier categories, or deserve separate mention for their unusual nature.

ISA is a static code analyzer that focuses on known code defects leading to vulnerabilities based on data fusion techniques to validate the results [84]. The tool combines the results from a number of known tools and applies simple reasoning on the data to validate them against each other. Although reported to increase the accuracy by cross validation, ISA's achievements are not widely cited and are not free from arguable contradictions. In case many tools have the same false positive, it will be identified by ISA as a true positive and a true positive might be identified as a false positive if it is cited by few detection tools. Due to the flexibility on the choice of detection tools, however, ISA can be applied to detect any type of code defect.

The 2008 dissertation by Csallner [36] presented the idea of the combination of over and under approximation analyses for testing. In the dissertation, over-approximated processes are reported as the processes having complete recall, and under approximated processes are the ones having complete precision.

Csallner and Smaragdakis [38] used test generation to refine the results of static analysis (abstract reasoning) to develop a tool named Check 'n' Crash (CnC). The approach used ESC/Java [51] to statically detect defects from source code and then validated the results using test cases generated via JCrasher [37], a tool developed earlier by the same authors. The two tools are combined together using a constraint solving technique that solves the constraints inferred from the ESC/Java output and provides input to the JCrasher. CnC is developed explicitly for Java code and is reported to detect a number of low level errors in arbitrary codes. The paper claims that the combination of the techniques outperform the same techniques used separately.

Almost at the same time, the idea of Csallner and Smaragdakis [38] had been used in reverse to establish a path pruning mechanism by Vipindeep and Jalote [157]. Instead of refining the outcome of static analysis by test cases, their technique used the data from testing to exclude the paths used in static analysis that might have relatively low possibility of errors [157]. The tool is effective in finding invalid references, initialization problems and null dereferencing.

The same idea of the combination of static analysis and test generation has been extended into aiding C program debugging by Cebaro et al. [22] through a framework named SANTE (Static ANalysis and TEsting). Their approach used the static analyzer Frama-C [52] and the structural test generation tool, PathCrawler [127].

Jiang et al. [76] explored the completely different approach on defects by analyzing clone related inconsistencies to find defects that might be introduced by cloning (i.e., duplicating code). The approach detects code clones and then extracts inconsistencies-of-interest to generate potential defect reports based on parse trees. However, in the same paper, the authors reported a false positive rate as high as 90%, attributed to some fundamental clone detection issues (e.g., same control constructs may not be clones, different drivers may introduce some clones). The assumption of "clone inconsistencies predicting bugs", which is one of the same that was used by CP-Miner [104] (another clone detection tool that directly influenced Jiang's approach), is also not beyond considerable doubt. The tool mentions context to describe the control construct of a particular code artifact [74].

4.1.11 Frameworks

Besides individual tools, a number of frameworks were developed to create tools for defects detection. SUDS [92] is a multi-stage defect detection framework that provides the testing engineers a mean to create code defect detection tools based on predetermined or custom built correctness models. SUDS work by four phases, namely, parsing, simplification, static analysis and instrumentation [92]. Although SUDS uses static analysis to generate its tools, it generates tools that are dynamically verifiable. SUDS itself does not detect any defects, rather creates mechanisms that do the detection.

The main notable technique used for SUDS is the source transformation technique [92]. It uses the parser from CTools [92, 40] and generates statements that are further simplified for analysis through the application of program slicing techniques. A "tainted propagation algorithm" is used to segregate the variables of interest and to observe their values, and thus providing an instrumented version of the source code that can then be run to find the values of interest.

Of the results reported in the paper that describe SUDS [92], it is applied on eight systems and found sixteen bugs with a few false alarms. An infrastructure like SUDS is a very powerful ally to testers, with the possible downside of code instrumentation that poses both a security risk and may not be effective in catching Heisenbugs. The effectiveness of SUDS was not found to be tested with large datasets.

A different approach was exhibited by Lu et al. [112] through PathExpander, an architectural support for improving the path coverage of the dynamic defect detection tools. The framework increases the coverage by executing the non visited execution paths in what the authors called a "sandbox", and thus is able to predict some of the defects that might exist there. However, the architecture can only resolve defects that are left undetected for the path coverage problem, and has no way of detecting those that are undetected for the value coverage problem [112].

Among the few works that address the scalability-accuracy trade-off, Parfait [33] is a multilayer framework to work on defect detection that utilizes multiple lists to explicitly improve the precision and recall of the outcome. The framework ensures both precision and scalability by combining a step-by-step approach in detecting "certainly defective", "possibly defective" and "safe statements" on C code.

Table 4.1: Properties of the Analyzed Tools

		Strategy			Path		Sensitivity				Requirements					Generality					Acc.		
#	Tech./Author	Static	Symbolic	Dynamic	Consideration	Processing	Path Sensitivity	Context Sensitivity	Inter-procedural	Intra-procedural	Code/Build	Annotation	Specification	Style	Structure	Languages	API / Frameworks	Source Code	Intermediate Repr.	Native Code	Precision	Recall	Comment
1.	ARCHER [163]	●	●	○	●	◐	◐	●	●	●	●	◐	○	○	○	⊙	○	◐	○	○	◐	◐	[163, 168, 96]
2.	Check n Crush [38]	●	○	●	◐	⊙	○	○	○	○	●	●	●	○	○	⊙	○	◐	◐	○	⊘	◐	[36]
3.	Chen [24]	●	○	○	●	●	◐	○	○	◐	◐	○	○	○	○	⊙	○	◐	○	○	⊘	⊘	
4.	CCI [77]	○	○	●	⊙	⊙	○	○	○	○	●	●	○	○	○	◐	●	○	○	●	●	⊙	
5.	DSD Crasher [39]	◐	○	●	⊙	⊙	○	○	○	◐	●	●	●	○	○	⊙	○	◐	◐	○	⊘	◐	
6.	ESC/Java [51]	●	○	○	◐	◐	◐	◐	●	●	●	●	●	◐	◐	⊙	○	●	○	○	◐	◐	[36, 38]
7.	Findbugs [71]	●	○	○	◐	◐	●	●	●	○	●	◐	○	○	○	⊙	◐	●	◐	○	◐	◐	
8.	Ganapathy [56]	●	●	○	◐	◐	◐	●	○	◐	●	○	◐	○	○	⊙	○	◐	○	○	⊙	◐	
9.	HOLMES [27]	○	○	●	◐	◐	●	○	○	○	●	●	●	○	●	⊙	○	◐	○	●	◐	⊙	
10.	ITS4 [154]	●	○	○	◐	◐	○	◐	○	◐	●	●	●	○	○	◐	○	◐	○	○	◐	◐	
11.	Jiang [74]	○	○	●	◐	⊙	●	●	○	○	●	●	●	○	○	⊙	●	●	○	●	●	◐	
12.	LCLint [49]	●	○	○	◐	◐	◐	◐	◐	○	●	●	◐	⊙	○	⊙	○	◐	○	○	⊙	⊙	
13.	Liblit [106]	○	○	●	⊙	⊙	○	○	○	○	●	●	○	○	●	●	●	●	○	●	●	⊙	
14.	Parfait [103]	●	●	○	◐	◐	◐	◐	◐	○	●	○	●	○	○	⊙	○	●	○	○	●	◐	
15.	Recon [113]	○	○	●	⊙	⊙	○	●	○	○	●	●	●	○	●	⊙	●	○	●	●	●	◐	
16.	Roychowdhury [138]	○	○	●	⊙	⊙	○	○	○	○	●	●	●	○	○	◐	○	○	◐	●	⊘	⊘	
17.	SAFE [57]	●	○	○	◐	◐	●	●	◐	●	●	○	●	○	○	⊙	○	●	○	○	◐	◐	
18.	SOBER [109]	○	○	●	⊙	⊙	○	○	○	○	●	●	●	○	◐	⊙	○	○	◐	●	◐	◐	
19.	SPLINT [48]	●	○	○	◐	◐	◐	◐	○	●	●	●	◐	○	○	⊙	○	●	○	○	◐	◐	
20.	Trainin [150]	●	○	●	⊙	⊙	○	○	○	○	●	○	○	○	○	⊙	●	●	○	○	⊙	⊙	
21.	UNO [68]	●	●	○	◐	◐	◐	◐	○	◐	●	○	○	○	○	⊙	○	●	○	○	⊙	⊙	[168, 29]
22.	Vrisha [166]	○	○	●	⊙	⊙	○	○	○	○	●	●	○	○	○	⊙	●	○	○	●	◐	⊙	
23.	Wagner [158]	●	○	○	●	◐	●	◐	◐	●	●	○	○	○	○	⊙	○	●	○	○	⊙	⊙	
24.	SRTA	●	●	○	●	●	◐	●	●	●	◐	○	○	○	○	●	○	●	○	○	●	●	Chapter 5

Strategy: ●= Used as the main technique, ◐= Used as auxiliary means, ○= Not used
Path: ●= All paths, ◐= Most paths, ⊙ = Minimum paths
Sensitivity: ●= Exclusively addresses, ◐= Implies, ○= Does not address
Requirements: ●= Requires, ◐= Does not require, but can use and derive benefits from, ○= No use
Code/Build: ●= Requires complete implementation, ◐= Able to use partial code , ○= No mention
Generality: ●= Considers all, ◐= Considers partly, ○= Does not Consider
Language: ●= General specification based, ◐= Multiple Language, ⊙ = Single Language
Accuracy: ●= $\geq$ 75%, ◐= $\geq$ 50%, ⊙ = $\geq$ 25%, ○= < 25%, ⊘ = Not enough data / undecidable

4.2 Defect Detectors Evaluation and Comparison

In the literature we analyzed, evaluation and comparison of defect detectors were done using one of the two ways. Most of the comparative studies developed benchmarks to evaluate the tools. In the second approach, either the benchmarks or other experimental methodology was used to compare a set of tools.

4.2.1 Benchmarks

Not many benchmarks exist in the field of defect detection. In their attempt to evaluate static analysis tools, Zitser et al. [168] used open source code as a benchmark for evaluation. Although their target was not to establish a benchmark, their approach constructed one. Lu et al. [112] constructed another benchmark for memory related vulnerabilities under the name BugBench.

Cifuentes et al. [32] proposed a benchmark suite to evaluate defect detection tools for systems developed in C. The benchmark, named BegBench, contains two sets of open source software to measure the accuracy and scalability of the defect detection software [32]. It was evaluated with the tools Parfait, Splint, Clang and UNO [32].

iBUGS, an automated benchmark extractor from development history, was constructed by Dallmeier and Zimmermann [43]. Instead of being a benchmark itself, the tool constructs benchmarks by extracting codes from existing codebases before and after fixing a defect (as identified by log messages) and uses them to construct the benchmark itself [43].

4.2.2 Evaluative and Comparative Studies

Shahriar and Zulkernine [141] classified static buffer overflow detectors over six characteristics, inference technique, analysis sensitivity, analysis granularity, soundness, completeness and supported language. Their assessment was on twelve tools that represented the state-of-the-art at present.

In a separate work, Zitser et al. [168] evaluated five static analysis buffer overflow detectors using open-source code as benchmarks. This study pointed out a number of issues in present systems, as was described in earlier sections in this Chapter. One particular contribution of this study was the representation of results using an ROC-type plot which was adopted in this dissertation for the presentation of results.

4.3 A Scenario Based Comparison of Defect Detection Tools

Table 4.1 summarizes the properties of the tools and techniques presented in this chapter, with a row dedicated to the later described SRTA (Symbolic Range Tuple Analyzer), the tool developed as a part of this thesis. The table presents the information in six different groups. ‘Strategy’ identifies the solution’s strategy in a combination of three major techniques, static, dynamic and symbolic analysis. The ‘Path’ group expresses the tool’s consideration and treatment to the set of execution paths. ‘Sensitivity’ identifies the tool’s analysis from the perspectives of path sensitivity, context sensitivity, inter-procedural analysis and intra-procedural analysis. The ‘Requirements’ group, in order, specifies the requirement of build-able or complete systems, requirement of annotation, and the requirement of specification, specific type or structure in the subject system. ‘Generality’ expresses the tool’s adaptability across different granularities. Finally, the ‘Acc.’ column specifies the Precision and Recall of the tools, as can be found in literature. A point to mention is, this table does not describe every tool mentioned in this chapter. It only described the ones directly comparable to SRTA, the technique developed in this research.

Table 4.1 can be used in four different ways. First, it provides a snapshot of a particular tool if read in a row-wise order. Second, it can provide the overview across particular perspectives, if read in a column-wise order. Third, it can provide a qualitative comparison of different tools under different perspectives. Fourth, it can identify the major issues and challenges present in the trade by analyzing the specific column values.

This section provides three scenarios that describe the use of the information summarized in Table 4.1

and Table 4.2. Instead of relying on hypothetical scenarios, the author's specific experience while doing the literature survey for this dissertation was expressed. The three scenarios describe the first three of the uses described in the previous paragraph. As the fourth use, identification of major issues and challenges, is specifically important for such a study, it is reported under a different section.

For the evaluation of the technique proposed in this thesis, different tools were required to compare the technique. Three different tools were planned for evaluation. The information presented in Table 4.1 and Table 4.2 was used to make the decision.

4.3.1 Scenario 1: Finding Tools Matching Perspectives

This thesis focuses on static analysis, and specifically, symbolic analysis. dynamic or hybrid analysis tools therefore do not make relevant candidates for comparison as the set of strengths and weaknesses for Static and Dynamic Analyses are often complementary to each other.

A look at the tools specified in Table 4.1 under the column group 'Strategy' specified the tools ARCHER, Chen's Tool, ESC/Java, Ganapathy's Tool, FindBugs, ITS4, LCLint, Parfait, SAFE, SPLINT, UNO, and Wagner's Tool as the potential candidates. Other tools could safely be excluded from the candidacy list for their strong reliance to dynamic analysis, which would make the comparison improper.

4.3.2 Scenario 2: Finding Information About a Tool

Scanning the Table 4.1 row-wise, it was found that ARCHER, Chen's Tool and Wagner's Tool considers all paths, same as SRTA. But only Chen's Tool processes all paths. On the other hand, Chen's Tool lacks the sensitivities required for SRTA, while ARCHER, ESC/Java, Ganapathy's Tool, FindBugs, LCLint, Parfait, SAFE, SPLINT, UNO and Wagner's tool provide more comprehensive treatments. Considering these facts, other tools were preferred over Chen's Tool for comparison.

All of the tools considered in this list required complete code, as expressed in Table 4.1. Therefore no advantage could be gained by selecting one over the other. But for the requirements of style and specific structure, ESC/Java and LCLint provide less compelling information. Other tools, however, do not have the extra burden of style and structure as ESC/Java and LCLint do. In consideration of these factors, ESC/Java and LCLint were not preferred over the rest of the tools for comparison.

Although SPLINT, FindBugs and Ganapathy's Tool require annotation that ARCHER, SAFE, UNO and Wagner's Tool did not, it was not considered a determining factor due to the existence of this requirement for many of the tools.

4.3.3 Scenario 3: Qualitative Tool Comparison

Considering the Table 4.1, it can be seen that, according to the literature, Ganapathy's Tool performs poorly in precision in comparison with other tools, although the recall performance is considerable. A further examination by Table 4.2 shows that Ganapathy's tool's defect coverage, in comparison with the others considered, is minimal. Same issue extends to Parfait. Considering these two facts, other tools were preferred over Ganapathy's Tool and Parfait.

The same factor of low precision and recall exists for Wagner's Tool and UNO, but UNO provides more defect coverage that Wagner's Tool. Table 4.2 also demonstrates SPLINT's strong association to multiple defect types, while ARCHER, and Wagner's Tool usually have weak association. Although SAFE's defect coverage is more numerous than the rest, the association to no defect class is strong. Considering these facts, FindBugs, SPLINT and UNO were considered as the final set of tools for comparison. The comparison was presented in Chapter 6, along with the details of experiments and evaluation.

4.4 Issues and Challenges

Despite the fact that a lot of efforts were indulged into the detection of defects, an equally large barrier of issues and challenges still remain. The generalized track for defect detection still has a lot of open problems to explore, as well as having scopes for improvement of present techniques to provide better results.

4.4.1 Defect Type Coverage

Table 4.2 shows the defect class coverage of the analyzed tools. For any defect type, a strong indicator is used if the tool explicitly aimed at that defect class, and does not exclude any defect type that has been found to be belonging to the class. A weak indicator is used if either the tool does not provide an explicit coverage to the class, or if it considers defect types to only part of the class.

As shown in Table 4.2, there are a number of defect classes that are focused heavily by the detection tools and thereare classes that are almost untouched. One of the heavily focused groups is the group for memory related defect classes that was covered by a number of tools (e.g., [163, 49, 50, 154, 57, 158, 56, 51, 36, 38]). Although most of the tools are vulnerability detectors, they often treat memory issues as vulnerabilities.

Apart from the concurrency problems that require special techniques to be employed, types requiring generalized techniques or extensions of present tools are still not covered much. For example, computation defects are investigated by only a few tools [154, 51, 36, 38], as well as those of data related ones [51, 36, 38], while Logical Defects stay mostly untouched.

Computation and data related defects largely fall into the categories where constraint checking can detect or infer the presence of errors. Thus a number of existing tools [163, 49, 50] that do not address these defects can be adjusted to work with them. Also, specific broad symbolic techniques [103] can be brought to work with these using their standard rule-set by defining rules for the types of errors.

The data from SRTA are presented in a shaded row at the bottom for comparison with other tools. As it is apparent from the table, SRTA provides a more comprehensive defect coverage than most of the tools analyzed in this chapter.

Table 4.2: Defect Coverage of Different Tools (with respect to the Taxonomy Developed in Chapter 3)

		Comp			Logic								Memory				Data					Sync.		
#	Tool/Author	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	
1.	ARCHER [163]	◐	○	◐	○	○	○	○	○	○	○	○	◐	○	○	●	○	○	○	○	○	○	○	[163, 168, 96]
2.	Check 'n' Crash [38]	◐	◐	○	○	○	○	○	○	○	○	○	◐	○	○	◐	◐	◐	◐	○	○	○	○	[36]
3.	Chen [24]	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	
4.	CCI [77]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	●	
5.	DSD Crasher [39]	◐	◐	○	○	○	○	○	○	○	○	○	◐	○	○	◐	◐	◐	◐	○	○	○	○	
6.	ESC/Java [51]	◐	◐	○	○	○	○	○	○	○	○	○	◐	○	○	◐	◐	◐	◐	○	○	○	○	[36, 38]
7.	FindBugs [71]	◐	◐	◐	◐	○	◐	○	◐	○	◐	○	◐	○	○	○	◐	◐	◐	○	○	○	○	[8, 8, 69, 71]
8.	Ganapathy [56]	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	
9.	HOLMES [27]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	
10.	ITS4 [154]	◐	○	◐	○	○	○	○	○	○	○	○	○	○	○	◐	◐	◐	○	○	○	○	○	
11.	Jiang [74]	○	○	○	◐	○	○	○	○	○	◐	○	◐	○	○	◐	○	○	○	○	○	○	○	
12.	LCLint [49]	○	○	○	○	○	○	○	○	○	○	○	●	◐	◐	●	○	○	○	○	○	○	○	[163, 49, 48]
13.	Liblit [106]	○	○	○	○	○	○	○	○	○	○	○	◐	◐	◐	◐	◐	◐	○	◐	○	○	○	
14.	Parfait [103]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	
15.	Recon [113]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	
16.	Roychowdhury [138]	○	○	○	○	○	○	○	○	○	◐	○	◐	○	○	◐	○	○	○	○	○	○	○	
17.	SAFE [57]	◐	◐	◐	○	○	○	○	○	○	○	○	◐	○	◐	◐	◐	◐	◐	○	○	○	○	
18.	SOBER [109]	○	○	○	○	○	○	○	○	○	○	○	◐	◐	○	◐	○	◐	○	○	○	⊙	⊙	
19.	SPLINT [48]	○	○	○	○	○	○	○	○	○	○	○	●	●	◐	●	○	○	○	○	○	○	○	
20.	Trainin [150]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	◐	
21.	UNO [68]	○	○	●	○	○	○	○	○	○	○	○	◐	○	○	●	○	○	○	○	○	○	○	[168, 29]
22.	Vrisha [166]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	◐	
23.	Wagner [158]	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	●	○	○	○	○	○	○	○	
24.	SRTA	●	●	●	●	●	◐	○	◐	●	●	○	●	●	●	●	◐	◐	●	●	◐	○	○	Chapter 5, Chapter 6

●= Strong Support, ◐= Weak Support, ○= No Support

4.4.2 Complete System Requirement

Although one of the strongest points in favour of static analysis is its ability to process incomplete code, this is often not utilized by the techniques. Prominent tools that utilize only static analysis [163, 51, 49, 48, 154] require complete systems to process, as apparent in the "Code/Build" column in Table 4.1. For any system to undergo a defect detection activity by these tools, the developer has to wait till a build-able complete system is at hand, forcing the detection to the later phase of development. Among the tools analyzed in this chapter, only SRTA and Chen's Tool skip the requirement of complete system.

4.4.3 Annotation

Tools to exploit dynamic analysis require source code instrumentation as a prerequisite for the application of the technique. Static analysis does not have the limitation of enforcing instrumentation, still a lot of static analysis tools rely on instrumentation or annotation. These instrumentations or annotations are often done by hand, involving significant human effort on the part of the developers. Often the developers of the tools report the significant overhead associated with the tool [38], which is supported by other researchers [163, 168].

Table 4.1 show the tool's annotation requirement under Column "Annotation". As it can be seen from the table, most tools require annotation as a requirement, while others consider it as supporting mechanisms. Only a handful of tools (Chen's Tool, Ganapathy's Tool, Parfait, SAFE, Wagner's Tool) exclude the requirement altogether. SRTA, the tool developed in this thesis, avoids the annotation requirement to provide for a more practical solution, as it can be observed in the shaded row of the table.

4.4.4 Accuracy

As presented under the columns Precision and Recall in Table 4.1, the precision and recall for most of the tools, especially those employing static analysis, are often in the order of 50%, with some performing in as low as in the order of 25%. In balancing the trade-off between scalability and accuracy, most tools often ensure scalability at the expense of accuracy [163, 158, 36], resulting in large number of false positives, and often having a large false negative rate.

4.5 Summary

This chapter provides an overview of the state-of-the-art with their comparisons with each other and with the technique developed in this research. The descriptions provided in this chapter is used to shape the development of SRTA, as described in Chapter 5. The chapter started by providing information on the tools and techniques categorized by their strategies, and then provided a qualitative comparison of the relevant ones. The tools and techniques were evaluated using the taxonomy developed in Chapter 3. The chapter then proceeded on describing a case study to demonstrate the use of the information contained in this chapter. Finally, the chapter described the prominent issues as were apparent from the information presented.

Chapter 5

Using Symbolic Range Tuples in Defect Detection

This chapter provides details on SRTA (Symbolic Range Tuple Analysis), the technique developed under this research to detect dissimilar classes of source code defects. SRTA builds on the concepts of defect similarity as outlined in Chapter 3 and uses the analyses provided in Chapter 4. This chapter begins by providing chapter-specific background information (Section 5.1) followed by a description of the propositions made by other researchers that drove SRTA's development (Section 5.3) and a description of the conceptual architecture of SRTA (Section 5.4). Next, the solution provided by SRTA, that utilized the propositions, was described (Section 5.5) along with the rationale behind specific design decisions. The Symbolic Domain, notations, rules and Symbolic Algebra associated with the solution are described next (Section 5.6). The chapter then continues with the details of application of SRTA for defect detection (Section 5.7). Finally, a prototype implementation of SRTA is described in brief (Section 5.13). This chapter answers the Research Questions RQ1 ("Can a specific abstraction provide sufficient means for detecting multiple dissimilar classes of defects?") and RQ2 ("To what length can such a technique go in terms of dissimilar classes of defects?").

5.1 Chapter-specific Background

This section provides the background for this chapter.

SRTA: Abbreviation for Symbolic Range Tuple Analysis. It is a dissimilar defect detection technique that was developed as the outcome of this research and is described in this chapter.

Path Envelopment: This term is used throughout this chapter to mean the actual expanse of the technique in probing the execution paths. A partial path envelopment is used to denote the situation where a technique does not process all possible execution paths across a program execution point, rather selects a subset of it. A Complete Path Envelopment is the situation where a technique considers each and every execution path that might have resulted in reaching a particular program point.

Path Summarization: The process of creating path summaries. Path summaries, in the context of this research, is the technique of creating a summarized representation of an entity's states in a program execution point, across all execution paths that have passed through that particular program execution point.

5.2 The Need for Detecting Dissimilar Defects

A single software project usually contains multiple classes of defects, some of which may belong to similar classes (e.g., different memory problems), and some in completely different classes (e.g., memory problems and computation errors). Regardless of their affiliations, even trivial defects are known to create complicate problems. One such case was the Mars Polar Orbiter Accident [100], the crash of a spacecraft that is attributed to a very simple error involving only units of measurement (NASA ground control team used the Metric Newton-Second as the unit of impulse, while the spacecraft's control software interpreted it as the Imperial Pound-Second unit, causing a discrepancy in its impulse estimation at a factor of 4.45) [146, 110]. Apart from the specialized Mars Orbiter Project, one other trivial defect caused human casualties in Canada and the U.S.A. by overdosing radiation patients with 100 times the intended value due to a unchecked keystroke sequence [101].

Considering an everyday software, this research found the popular browser Mozilla Firefox alone to be a container of twelve different classes of defects (Table 6.7 in Chapter 6). The classes belong to at least four different dissimilar groups, even under the most liberal estimation of similarity.

To provide a complete detection coverage on software projects, a software management team has two alternative solutions. First, a set of different tools providing specialized coverage to different defect types can be applied. This technique often fails to guarantee high quality software, as specialized tools focus more on scalability than accuracy (e.g., [163]). If only high accuracy tools are chosen, being originated in different sources, the tools still differ in their input specifications and reporting. Using the strategy effectively thus requires specialized and dedicated manpower, increasing project overhead.

The second approach is to use one of the available industry tools that provide coverage over different defect types (e.g., Coverity SAVE [35], Grammatech CodeSonar [63], Polyspace [129]), that, in addition to being expensive [86, 85, 64], require high configuration of the experimental environment to be able to conduct their massive analyses (as reported in [29]).

If multiple dissimilar defects can be detected by a tool with reasonable accuracy, it can provide a substitute to the multi-tool strategy of the first approach described above, as it is already provided by the tools mentioned for the second approach. If such a tool can retain its accuracy and scalability using a single simplified model, it should be able to reduce its internal structural complexity, thus relieving the requirement of high-resource experimental environments. Using such a tool shall be able to provide the benefits of the second approach, without the associated drawback of resource intensive requirements, and probably without the high acquisition cost. This particular motivation was the driving force behind SRTA.

5.2.1 Defect Classes Revisited

To make this chapter self-contained, a summary of the Taxonomy described in Chapter 3 is provided in this section in Table 5.1. The table does not contain examples or details of the defect classes. A summary of the identifying property is mentioned along with defect class identifier and the defect class's descriptive name.

Table 5.1: Summary of Defect Classes Described in Chapter 3

#	Class	Name	Description
1.	C1	Value Representation Defect	An entity is assigned a value it is unable to represent.
2.	C2	Value Offset Defect	An entity's value is off by a specific constant.
3.	C3	Undefined Outcome	An operation's outcome becomes non-determinable.
4.	L1	Improper Checks	Absent, or wrong, check at any point in code.
5.	L2	Improper Terminal Condition	Absent, or wrong, condition for loop or recursive function termination.
6.	L3	Wrong Operation	Using one operation / operator in place of another.
7.	L4	Flaws in Algorithm	Any error in algorithm.
8.	L5	Performance Issues	Presence of any code structure that could be replaced with a better performing one.
9.	L6	Improper Exception Handling	Not handling proper exception.
10.	L7	Improper Control Flow	Creating a control flow improper for the context.
11.	L8	Design Non-conformance	Deviation from design / algorithm.
12.	M1	Invalid Memory Reference	Accessing a memory location that should not be accessed.
13.	M2	Improper Deallocation	Deallocating wrong entity, or deallocating in the wrong way.
14.	M3	Memory Leaks	Not releasing allocated memory.
15.	M4	Over / Underflow	Accessing a memory out of the defined boundary.
16.	D1	Interface Mismatch	Using mismatching data types / other specifications in interfaces
17.	D2	Data Mismatch	Using mismatching data values in interfaces.
18.	D3	Improper Input Validation	Missing, or making error in, input validation.
19.	D4	Missing / Extra Inputs	Self-explanatory.
20.	D5	Improper Abstraction	Improper representation of an entity (not the value).
21.	S1	Prohibitive States	Multi-process situation that halts execution and cannot be resolved.
22.	S2	Improper Sequencing	Multi-process situation with wrong sequencing of events.

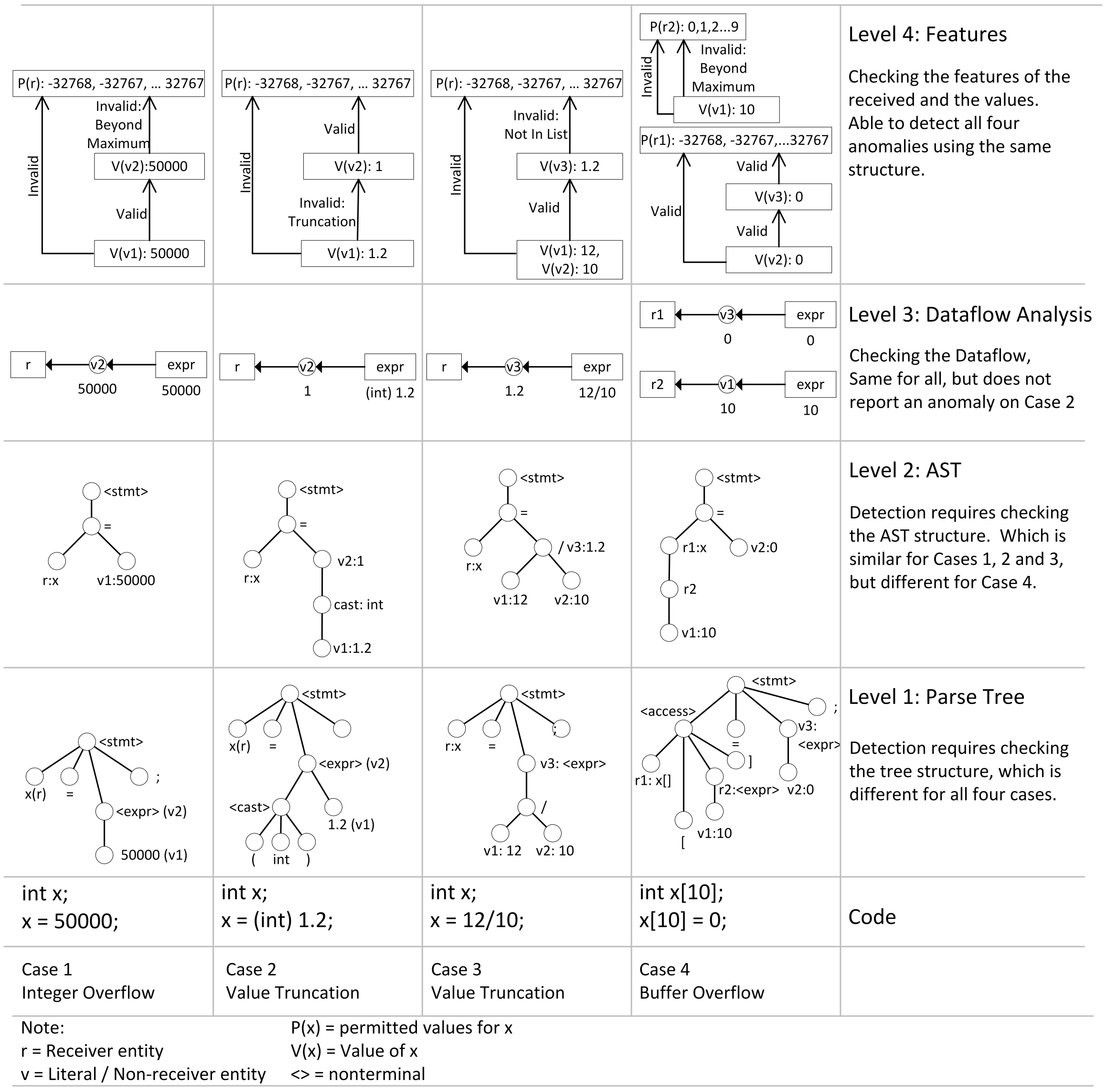
5.2.2 Single Model Representation of Dissimilar Defect Classes

Defect features simplify over abstractions. The higher one goes in the abstraction levels, the more simplified the feature set becomes. However, more abstraction also specifies less details - which limits the detection capabilities. If a specific abstraction can be provided over the source code to represent the entities in a form that can express multiple defects under the same structure, while preserving sufficient information for their detection and distinction, dissimilar defect detection from a single model becomes a reality.

Four illustrative cases, taken from simple examples, are shown in Figure 5.1. Of the four cases, the first three belong to one class of defect, the value representation problem, and the fourth belongs to a completely different class, buffer overflow. The four levels of abstraction show how the features converge over higher abstractions, facilitating the employment of single model.

The source code, including its entire token sequence can be represented using a Parse Tree (PT), as it is done for all software building processes. However, the source code contains defect instances, not defect types or classes. Therefore, detecting every single instance from a Parse Tree would require specialized treatment for different defect instances. In Figure 5.1, Case 2 and Case 3 are two instance of the same defect, although their parse trees differ. Any pattern matching algorithm that works on parse trees, unless it incorporates customized inference modules for such cases, will treat the two instances as two different defects. Although the trees have common subtrees, it cannot be used for the inference due to the fact that there are different non-terminals involved in the mismatching part of the trees.

The Abstract Syntax Tree, or AST, provides an abstract mechanism over the Parse Trees (which, in



Case 1 Integer Overflow	Case 2 Value Truncation	Case 3 Value Truncation	Case 4 Buffer Overflow	

int x;
x = 50000;
	int x;
	x = (int) 1.2;
		int x;
		x = 12/10;
			int x[10];
			x[10] = 0;
				Code

Level 4: Features

Checking the features of the received and the values. Able to detect all four anomalies using the same structure.

Level 3: Dataflow Analysis

Checking the Dataflow, Same for all, but does not report an anomaly on Case 2

Level 2: AST

Detection requires checking the AST structure. Which is similar for Cases 1, 2 and 3, but different for Case 4.

Level 1: Parse Tree

Detection requires checking the tree structure, which is different for all four cases.

Note:
r = Receiver entity
v = Literal / Non-receiver entity
P(x) = permitted values for x
V(x) = Value of x
<> = nonterminal

Figure 5.1: Representation of Four Defects over Four Levels of Abstraction

contrast, are often called the Concrete Syntax Trees), although it is not much different than a parse tree. ASTs can generalize the defect instances to defect types, although cannot extend to different defect types represented under the same defect class. In the figure, Cases 2 and 3 are represented by different tree structures, but the information can be obtained from either one as the differing parts of the sub-trees do not contain any non-terminal values, making the non-matching part deterministic and thus inferable.

Moving up one more step on the abstraction, if a dataflow module is used on top of the AST to comprehend the context of the information represented by the AST, different defect types corresponding to the same context can be generalized into one unit, effectively summarizing a defect class. If an analyzer performs at this level, it is able to detect a single, or a few related, defect classes using a single model or technique, although cannot generalize to dissimilar classes. In the figure, Cases 1, 2 and 3 result in identical dataflow anomalies each involving , although the Case 2 defect cannot be detected by dataflow analysis because the change in data occurred before the assignment to the receiver.

If the representation can be abstracted even further, then the defect classes can be represented by a set of features. At this point, the representation problem reduces to an abstract feature set, instead of the instance, type or class-specific representations of less abstract methodologies. If the source code's representation can be converged into the same abstract feature set, this set can be used as the model for the detection. The figure shows the situation on Level 4, where all five situations of the four example cases are represented using the same structure. SRTA utilizes this concept.

5.3 Existing Propositions

SRTA is built on five major propositions that were published over a period from 1995 [44] to 2011 [60].

Proposition 1: Source Code Contains Most of the Defects

Studies show that source code alone contains the origination and exhibition of 75% of the defects existing in software systems [99]. For the defects that are not originated in source code, often the footprint is contained in the source [44].

Proposition 2: Dissimilar Defects Show Similar Characteristics over Arbitrary Abstraction

Virtually all defect taxonomies are built on this concept that different defects can be expressed through common properties. The work by DeMillo and Mathur [44] used this fact to use the grammar as an abstraction to categorize defects. Although their work is not directly related to this research, differing from objectives, processes and experiments, it established the grammar based defect generalization principles that are directly utilized by this research in defect detection.

A 2010 study by Le and Soffa [96] has utilized the same principles, although through different realizations and procedures, into detecting multiple similar classes of defects using a simplified representation.

Proposition 3: Summarization Techniques can Alleviate Path Explosion Problems

As proven by Godefroid et al. [59, 60], compositional symbolic analysis effectively alleviates path explosion problems associated with static and symbolic analyses. In a later work, Li et. al. [103] used the summarized ranges in detecting buffer overflows through a tool named Parfait. Their argument proved the general technique of summarization to be practical and effective when analyzing large systems.

Proposition 4: Path Summarizations Preserve Defective Path Information

As demonstrated by Blume and Eigenmann [15, 14] and later applied by Wagner et al. [158] and Xie et al. [163], it is possible to convert code features into numerical ranges that can be analyzed to infer defects. In a recent study, Le and Soffa [93] has proven that, if a given program point contains a defect, any path traversing through that point contains properties relevant to that defect. The general argument of Le and Soffa [93] was against the use of summaries due to the fact that summaries do not retain enough information for effective detection. Recognizing this argument, this research used multiple summaries to resolve this issue.

Proposition 5: State-Space Holds Information for Defects

Le and Soffa [94, 95] have demonstrated that "faults exhibit locality". A generalization of this statement can be used to infer that defects exhibit features that are distinguishable in their local context.

5.4 An Overview of Symbolic Range Tuple Analysis

SRTA, or Symbolic Range Tuple Analysis, was aimed to be general, scalable, practical and accurate. To achieve generality, SRTA was designed with provisions for as much soft-coded specifications as possible. Figure 5.2 shows the conceptual structure of SRTA. In the figure, the external data (i.e., Source Code, Specifications and Grammar) are shown using a document symbol. The dotted rectangle denotes the boundary of SRTA and the solid rectangles denote the three functional phases, with their specific tasks listed inside.

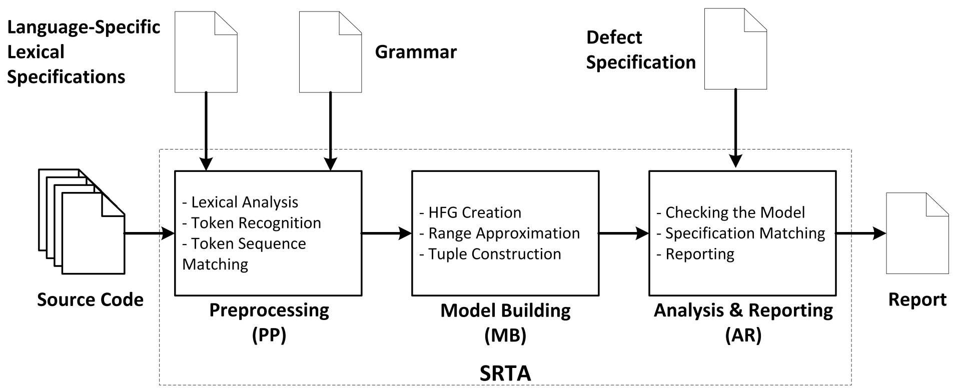


Figure 5.2: Conceptual Architecture of SRTA

For the analysis of a software system, the input to SRTA is the collection of source files that make up the software system. No extra information on the system or annotation is required. SRTA is composed of three sequential phases, shown in Figure 5.2.

5.4.1 The Preprocessing Phase

The first phase, 'Preprocessing', prepares the input data stream for processing by SRTA. This phase is not to be confused with the C/C++ Preprocessing, although the C/C++ Preprocessing is a part of SRTA's preprocessing phase. The functional tasks for the preprocessing phase is to tokenize the input according to the Language-Specific Lexical Specifications and to match the sequence of tokens into grammar patterns of interest, which is recognized from the Grammar supplied as external data. Both the Lexical Specifications and Grammar were incorporated as external components to ensure SRTA's applicability over different programming languages.

5.4.2 The Model Building Phase

The next phase is named 'Model Building'. In this phase, SRTA analyzes the token sequence resulting from specific grammar patterns, and creates a Hybrid Data and Control Flow Graph (HFG). The HFG is used to approximate the symbolic ranges SRTA uses. The Range Tuples, which are the main model components for SRTA, are inferred in this phase. This phase outputs the internal model for SRTA which is described in details in later sections of this chapter.

5.4.3 The Analysis and Reporting Phase

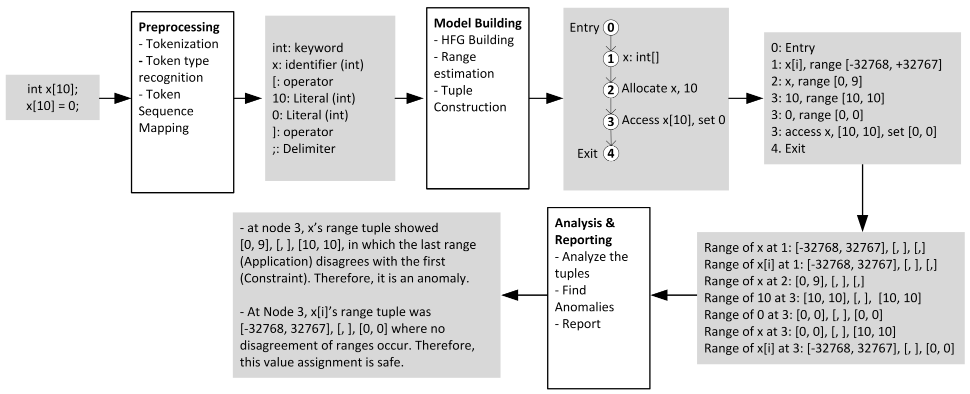
The third, and final, phase, named 'Analysis and Reporting', analyzes the information obtained from the model with respect to user supplied specifications on defects, and reports the outcome.

Later sections of this chapter details the model, the ranges, inference procedures, tuple description and the defect specifications, along with the description of a prototype to implement the concepts of SRTA.

5.4.4 Example Workflow

To demonstrate how SRTA works, the fourth case as represented in Figure 5.1 is used. This example is included in this section with the intention to provide an overview of SRTA's working procedures only, exact technical details and model specifications are provided later in this chapter.

Figure 5.3 shows the summarized workflow. In the preprocessing phase, SRTA breaks down the source code into tokens, determines the types of identifiers, and determines values for the literals.



```
int x[10];
x[10] = 0;
```

```
int: keyword
x: identifier (int)
[: operator
10: Literal (int)
0: Literal (int)
]: operator
;: Delimiter
```

```
0: Entry
1: x[i], range [-32768, +32767]
2: x, range [0, 9]
3: 10, range [10, 10]
3: 0, range [0, 0]
3: access x, [10, 10], set [0, 0]
4. Exit
```

```
Range of x at 1: [-32768, 32767], [, ], [,]
Range of x[i] at 1: [-32768, 32767], [, ], [,]
Range of x at 2: [0, 9], [, ], [,]
Range of 10 at 3: [10, 10], [, ],  [10, 10]
Range of 0 at 3: [0, 0], [, ], [0, 0]
Range of x at 3: [0, 0], [, ], [10, 10]
Range of x[i] at 3: [-32768, 32767], [, ], [0, 0]
```

- at node 3, x's range tuple showed [0, 9], [,], [10, 10], in which the last range (Application) disagrees with the first (Constraint). Therefore, it is an anomaly.

- At Node 3, x[i]'s range tuple was [-32768, 32767], [,], [0, 0] where no disagreement of ranges occur. Therefore, this value assignment is safe.

Figure 5.3: Example Workflow of SRTA Over a Simple Example

On the next phase, model building, SRTA builds a hybrid data and control flow graph (HFG) that represents the flow in the code. Using this graph, and the external specifications provided, SRTA assumes the range of values associated in each execution point, which is represented by one or more nodes in the graph. In this particular example, integers were assumed to be 2-byte entities. In every node, all possible values for an entity is considered. When x was declared (before initialization) at Node 1, its value was encompassing the entire range possible. At node 2, when the allocation was done, the range was restricted to [0, 9].

Using the HFG's flow information, SRTA constructs a 3-tuple of ranges, signifying the absolute possible range of values, the imposed (through external checks) range of values, and the applied range of values for an entity. In this example, there was no imposed range (they require conditional branches). The other two ranges were specified.

In the analysis phase, an anomaly is detected when a disagreement between ranges is found. In this case, at Node 3, x was accessed with a range of values [10, 10], where its permitted range was [0, 9], thus indicating an anomaly. On the other hand, the single element of x, being an uninitialized integer, has the permitted range [-32768, 32767] and was accessed using the range of values [0, 0], which fall within the permitted range. Therefore it was marked as safe.

Later sections provide the technical details of mathematical approximations, tuple description, tuple formation, range estimation and the detection procedure.

5.5 Solution Description

SRTA detects dissimilar defects by analyzing a single model. The model is built using symbolic values obtained using static analysis, through the application of path summarization and complete path envelopment techniques in the form of a 3-tuple containing symbolic ranges.

5.5.1 Rationale

In detecting dissimilar defects, the first problem arises in defining what defect similarity is, and which defects are similar or dissimilar under what conditions. Chapter 3, through the established framework, attempts to define defect similarity. The proposition that different defects can be similar with respect to arbitrary abstraction (Proposition 2), provides a possibility to model dissimilar defects into a simplified construct that can be analyzed for the detection.

Source code is selected for the processing because of the fact that it is the container of most defects (Proposition 1). As source code exhibits strict formal constructs, it is possible to apply automated techniques in analyzing source code and to find the defects. Moreover, the defects that are uncovered in earlier phase of development often require lower fixing overhead. Also, as dynamic analysis, being the other possible choice, cannot analyze paths that are not taken during the execution of the software, it often has high false negative rates for the defects that exist in the non-taken paths. Static analysis can uncover defects in all paths, thus resulting into lower false negative rates. Considering the advantages, and the objectives of the research, static analysis was chosen as the proper technique to be adopted.

Defects often exhibit themselves through the interactions of different code artifacts. Detection of defects often requires information on the inter-relation of different artifacts, as well as on the order and probability of invoking particular statements. It is difficult for a pure static analysis to analyze these information. Symbolic analysis aids in such situations by simulating execution or by inferring constructs that provide the information or approximations of runtime behaviour. To attain the objectives of this research, symbolic analysis was thus chosen as the technique of selection.

Usually the path explosion problem is attacked using two methodologies, exclusion of infeasible paths [121, 147, 167] and path summarization [31]. Exclusions are able to reduce the search domain for the analyzer to a great deal and are employed in different flavours, but they suffer from very prominent drawbacks. First, the general solution to the problem of finding infeasible paths is undecidable [121, 94] - thus the best outcome is bounded by a good approximation only. Second, often the required information is localized in statements under particular context [147], and are not decidable through pure static analysis. Third, the number of included paths, regardless of the soundness of the exclusion approximations, increases with the increase of total available program paths and puts a practical limit on the scalability.

Summaries do not exclude paths, but the specific summary technique may drop information required to detect the defects. Despite their lossy representation, summarizations are effective techniques to counteract path explosions (Proposition 3), because, as the set of paths is summarized, the size of the summary set does not increase with the increment of paths. Additionally, path summarizations can be devised to contain required information if the objective of summarization is defined (Proposition 4). For SRTA, the objective was predefined, and the summaries were constructed of multiple parts.

5.5.2 Statement Ordering

In SRTA, symbolic range tuples are used to model the state-space. The components (i.e., ranges) of the tuples are derived from the source code. Statement orders were used to model the relations among the components of the tuple, and among different tuples. The statement ordering was determined with the following principles

(i) Statement orders have arbitrary start points.

(ii) From the start point, statements have an ascending integer number associated with them to denote their location in the code.

(iii) In case of a procedure call, the statements continue inside the procedure code from the calling point, and return to the called point after completing the procedure.

5.5.3 State-space Modelling

A software's states are the different instantaneous situations of the software along its execution. A state-space is the collection of all states the software may encounter in its course of execution. The concepts related to state-space, and different modelling of the state-space, are frequently used in type systems and model checking. The same concepts for software state-space can be applied to individual entities like variables or constants.

For a primitive entity (e.g., an integer variable), the states are different values the entity may assume. For example, for a one byte signed integer value, the state space consists of all values from -128 to +127. States

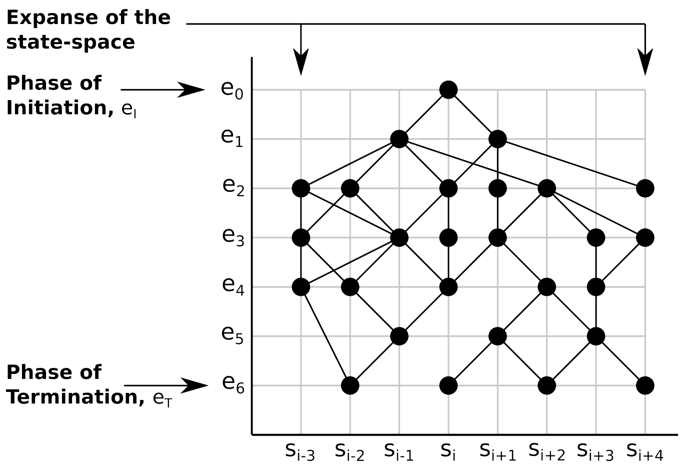

Expanse of the state-space
Phase of Initiation, e_I
Phase of Termination, e_T
e_0
e_1
e_2
e_3
e_4
e_5
e_6
s_{i-3} s_{i-2} s_{i-1} s_i s_{i+1} s_{i+2} s_{i+3} s_{i+4}

Figure 5.4: Generic State-space for an Atomic Entity

change with execution phases, where an execution phase is defined as a set of instructions that perform any change in the software or accesses any entity in any way. The exact machine level details of an execution phase may vary with different levels of abstraction but for the current analysis, considering only the change inducing instruction(s) as a single execution phase is sufficient (that is, more than one variable declaration statements may comprise to a single execution phase). At any execution phase, an entity has exactly one state out of a set of possible states.

Figure 5.4 shows a generic state-space for an entity. Execution phases are defined as $e_0, e_1, e_2 ...$, with e_0 being the initial state and $e_1, e_2, e_3, ...$ being subsequent execution phases. The states for the identifier are denoted as $s_{i\pm n}, n \geq 0$, with s_i as the initial state. At the initial phase of execution, e_0, the state of the identifier is at s_i. As execution progresses, at the first phase, e_1, depending on the situation in the software, the identifier may have any one of the states s_{i+1}, s_{i-1}. If the identifier was at state s_{i+1} in e_1, it may assume any one of the states s_i, s_{i+1}, s_{i+4} in e_2, and in case it was at state s_{i-1}, it may have any one of the states $s_{i-3}, s_{i-2}, s_i, s_{i+2}$ at e_2. The execution thus proceeds on, if the identifier is at any specific state s at an execution phase, in the next execution phase it either remains there, or has a transition to any of the states connected by lines with s. The range $[s_{i-3}, s_{i+4}]$ thus encompasses all values the identifier may assume over the course of the execution. The 'Phase of Initiation' is the execution phase where the entity comes into being - through declaration or allocation. The 'Phase of Termination' is the execution phase where the entity ceases to exist.

A few properties of the state-space becomes apparent from the diagram. First, when increasing values of i denote subsequent execution phases, there can only be a transition from phase e_i to phase e_{i+1} and not the opposite. The reason for this phenomenon is, if a defect occurs in a particular statement, there may not be a decidable relation of the defect with an earlier version of the same statement. To account for this fact, the execution is modelled as linear and not repetitive. For a recurrent structure that runs n-times and induces some change in an entity, there are n execution phases arranged in a linear construct where the transition to

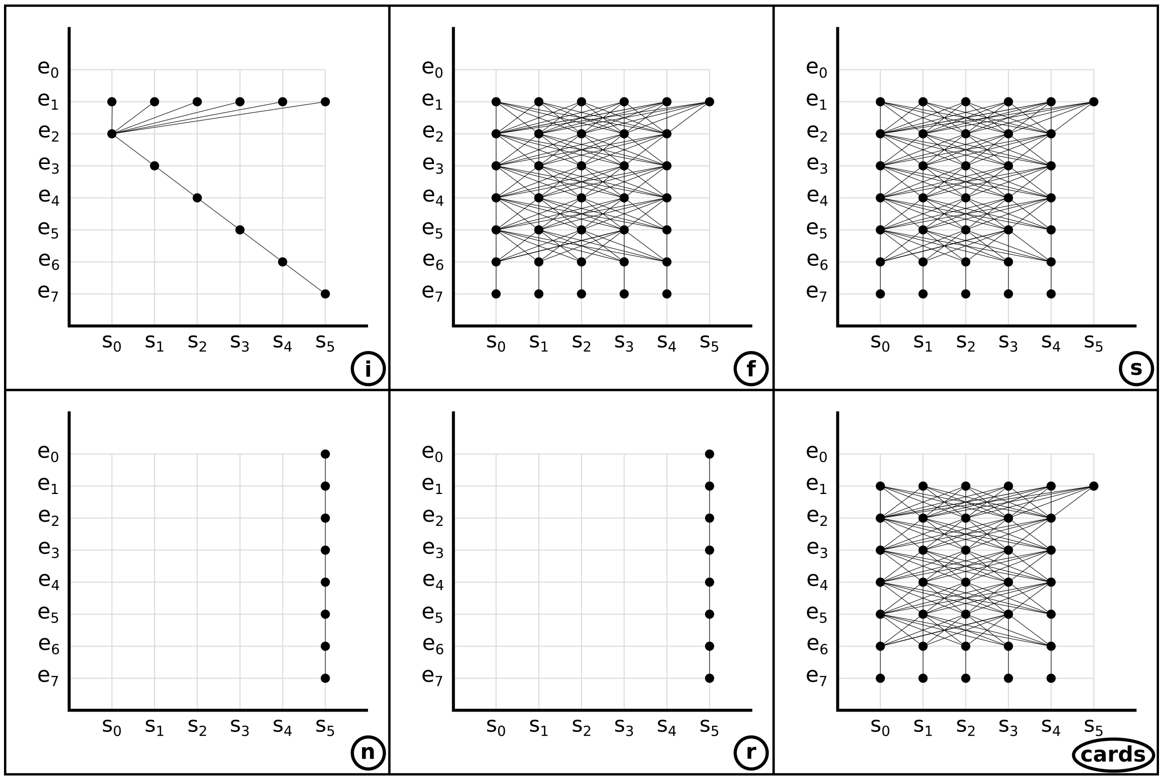


Figure 5.5: State-space for Entities in Listing 5.1

a new state depends only on the present state of the entity without any dependency on the previous states. As execution does not roll backwards, it is not possible for a state to go to another state in a previous phase although it might assume one of the previously assumed states in a later execution phase.

The second property is that, a transition from execution phase e_i occurs only to execution phase e_{i+1}, and not to any later state. This phenomenon realizes that if an entity is not changed by an execution phase, unless it is destroyed, it retains the same state it was in before proceeding into that execution phase and it does not become stateless or undefined. For the same reason, if an entity is created in e_I and destroyed in e_T, for every intermediate execution phase $e_i, I \leq i \leq T$ it has at least one transition. For any transition, the destination state may be the same as the state of origin (although under a different execution phase), or it may be a different state.

A more specific example is shown in Figure 5.5, following the Code Listing 5.1. The figure shows the state-space for entities n, r, i, f, s and $cards$ in the function $Shuffle()$ under the specified calling context. In the subfigures under Figure 5.5, execution phases are marked downward on the vertical axis and states, i.e., the values of the identifiers, are on the horizontal. The initial execution phase, e_0, is the invocation of the function. The next execution phase, e_1 includes the variable declaration statement as it does not change any other value in the statement. e_2 to e_6 includes the loop iterations, with e_7 marking the return statement. The states are formed using different possible values of the identifiers - that is, in state s_b entity a assumes the value b.

The constants n and r were created at e_0, when the function was entered and their values did not change over any execution phase. Both entities retained the same state throughout all execution phases, as shown in Figure 5.4(n) and Figure 5.4(r). The entities i, f *and* s were declared at e_1, but were not initialized. Their initial state can be any of the possible ones. For simplicity, the figure does not include states beyond s_6 but in reality, the initial states for i, f and s span the range of all possible states for the int-type (i.e., a total of 65536 states for 2-byte integers). In e_2, the value i, regardless of its state at e_2 moves to state s_0 due to the initialization inside the loop and steadily moves to the next states over the next execution phases. The values f and s, in execution phases e_2 to e_6, may assume any state from the set $\{s_0, s_1, s_2, s_3, s_4\}$ in any of the phases.

Listing 5.1: Example Code to Demonstrate the State-space

```
void Shuffle(int cards[], const int n, const int r)
{
  int i, f, s;
  for(i = 0; i < r; i++)
  {
    f = rand() % n;
    s = rand() % n;
    //Swap cards[f] and cards[s]
  }
  return;
}

int main(void)
{
 int cards[5];
 ...
 Shuffle(cards, 5, 5);
 ...
}
```

For SRTA, the state-space is modelled as three ranges signifying the three situations that can occur in the state-space. Usually, a state-space is conceptually modelled over a two dimensional plane with one direction denoting the states and the other denoting the execution phases that result into that state, as shown in Figure 5.4 and Figure 5.5. In SRTA, however, only summaries over execution paths are used, therefore a single range can signify the states over all execution paths. The three ranges are used to characterize the absolute allowed value of the state-space (the Constraint), any implied boundary within the state-space (the Resolution) and the actual state space-covered by an entity (the Application) in the course of the software being analyzed. Details on the tuple are provided later.

Considering the example code Listing of 5.1, the array *cards*, the absolute boundary for the state-space for the symbolic array indices is $[0, 4]$. When this array is passed into the function, the applied values for the state-space become the values bounded by $rand()\%n$, which is $[0, n-1]$, or, in this context, the range $[0, 4]$ that coincides with the absolute range. If, in the declaration of cards, its size was set to less than 5, or if n was set as greater than 5 in calling the function, reachable states for the array would have contained states that are invalid in its declaration - indicating a problem in the code.

The three-tuple representation of the state-space is applicable to atomic entities. For collective entities, each member of the collective entity contains its own state-space, with an additional state-space dedicated to the collective entity itself. The state-space for composite entities is a collection of state-spaces for the different collective and atomic entities it may contain.

5.6 The Symbolic Domain

This section describes the symbolic domain that SRTA operates on. The descriptions of the range tuple, ranges, their properties and interaction are provided.

5.6.1 Notations and Specific Values

To clearly convey the ideas, a set of specific notations were adopted from programming languages and mathematics. Although most of them were used in congruence with their conventional meanings, some of the symbols have been customized according to the need of this dissertation. The notations were adopted for specific values and operations. Table 5.3 provides a summary of the notations and specific values.

Absolute Limits

Two symbolic absolute limits have been used in the expressions. The limits, $-\infty_{type}$ and $+\infty_{type}$ denotes the theoretical minimum and maximum range of values accommodated by an entity that belongs to *type*. The $+$ and $-$ in front of the notation are not used strictly to denote values greater or less than zero, rather they are used to denote the position of one with respect to another (i.e., they denote the maximum and minimum of the range, both may have the same sign in their absolute values).

For any kind of literal (i.e., constant value), both absolutes are equal to the value of the literal. Table 5.2 shows the absolute limits for a few atomic and collective data types.

Symbolic Value and Set of Values

A Symbolic Value is a value assigned to an entity through a symbolic analysis or inference procedure. The value may or may not be the actual value that occurs in the entity during execution. It may or may not be the value assigned to the entity in its static context.

A Set of Values for an entity is the set of all possible symbolic values that could have been assigned to that entity in any point of execution.

Table 5.2: Examples of Absolute Limits for Different Entities

#	Type	Entity	Size	$-\infty$	$+\infty$	Comment
1.	char	Atomic	1 byte	-128	+127	
2.	unsigned char	Atomic	1 byte	0	+255	
3.	int	Atomic	2 bytes	-32768	+32767	
4.	unsigned int	Atomic	2 bytes	0	+65536	
5.	int[n]	Collective	N/A	0	n-1	
6.	const int n = 5;	atomic	2 bytes	5	5	

Table 5.3: Notations, Values and Symbols

#	Symbol	Example	Explanation
Sets, Values and Generic Symbols			
1.1.	$\pm\infty_{type}$	$+\infty_{int}$	Absolute limits for values under the type. Unconventional.
1.2.	ϵ, E		Symbolic expression and set of all symbolic expressions. Unconventional.
1.3.	v, V		Generic single symbolic value and set of symbolic values.
1.4.	R, R_i		Generic range of values.
Operations			
2.1.	$\leftarrow$	$R_1 \leftarrow R_2$	Assignment. Conventional.
2.2.	$+, -, *, /, \%$	$R_1 + R_2$	Arithmetic operations. Conventional.
2.3.	$<<, >>$	$R_1 << V$	Bit shift. Conventional
2.4.	$\cup$	$R_1 \cup R_2$	Union. Conventional.
2.5.	$\cap$	$R_1 \cap R_2$	Intersection. Conventional.
Relations			
3.1.	$\in$	$R_1 \in R_2$	Set inclusion. True if every elements of R_1 is also an element of R_2.
3.2.	$\top$	$R_1 \top R_2$	Containment. True if R_1 is contained-over-maximum in R_2.
3.3.	$\perp$	$R_1 \perp R_2$	Containment. True if R_1 is contained-over-minimum in R_2.
3.4.	I	$R_1 \mathrm{I} R_2$	Containment. True if R_1 is completely contained in R_2.
3.5.	$\mid\ \mid$	$\|R\|$	Cardinality. Conventional.
3.5.	$\|\ \|$	$\|\|R\|\|$	Cardinality of subrange. Unconventional.

Symbolic Expression and Set of Expressions

A Symbolic Expression, ϵ is an expression that uses symbolic values instead of the actual values. The set of expressions, E is the set of all symbolic expressions in the context of the entire software.

5.6.2 The Range Tuple

The range tuple is defined to reflect the model of the state-space and contains three parts - namely, the Constraint, the Resolution and the Application. That is, for any entity x, at statement i, there is a range tuple,

$$T_x(i) = \begin{cases} \{C_x(j), R_x(k), A_x(i)\} & \text{if x is an atomic or collective entity} \\ \bigcup_{\forall y}\{T_y(i) | \text{y is a member of x}\} & \text{if x is a composite entity} \end{cases} \tag{5.1}$$

Where,

$C_x(j)$ = the Constraint, or absolute boundary of x's state-space as defined on statement j.

$R_x(k)$ = the Resolution, or the implied restriction on x's state-space as defined on statement k.

$A_x(i)$ = the Application, or the actually applied value on x on statement i.

Of the statement numbers, following relations can be obtained,

$j \leq i$, because the Constraint is fixed at the time of declaration / reallocation.

$k \leq i$, because checks and bounding constructs always work before the entity is accessed.

To elaborate the ranges further, the Constraint can be expressed as a continuous range of a minimum and maximum. The elaboration of Constraint thus is,

$$C_x(j) = [min(C_x(j)), max(C_x(j))] \tag{5.2}$$

where, $max()$ and $min()$ denote the maximum and minimum value for the range, respectively.

For the Resolution, the range may not be continuous, rather may contain disjoint subranges. The Resolution can be expressed as,

$$R_x(k) = \{R_{x1}(k), R_{x2}(k)...R_{xn}(k)\} \tag{5.3}$$

where every single $R_{xi}(k)$ is a disjoint subrange of the values. And for every $R_{xi}(k)$

$$R_{xi}(k) = [min(R_{xi}(k)), max(R_{xi}(k))] \tag{5.4}$$

If R_x contains more than one subrange, the extremes of R_x is defined as the extremes of all the subranges. For the lowest extreme,

$$min(R_x(k)) = min(\forall_{(R_{xi}(k) \subset R_x(k))}(min(R_{xi}(k))) \tag{5.5}$$

and the highest extreme becomes the highest of the highest extremes of all subranges, i.e,

$$max(R_x(k)) = max(\forall_{(R_{xi}(k) \subset R_x(k))}(max(R_{xi}(k))) \tag{5.6}$$

For the Application, the same issues of the Resolution applies, thus

$$A_x(i) = \{A_{x1}(i), A_{x2}(i)...A_{xn}(i)\} \tag{5.7}$$

Where, every $A_{xi}(i)$ is a subrange of the values,

$$A_{xi}(i) = [min(A_{xi}(i)), max(A_{xi}(i))] \tag{5.8}$$

The lowest and highest extremes of A_x also follow the same definitions as those of R_x, that is,

$$min(A_x(i)) = min(\forall_{(A_{xi}(i) \subset A_x(i))}(min(A_{xi}(i))) \tag{5.9}$$

$$max(A_x(i)) = max(\forall_{(A_{xi}(i) \subset A_x(i))}(max(A_{xi}(i))) \tag{5.10}$$

By definition, the Resolution is simply a restriction imposed on the Constraint. If an entity is not used inside the bounding block of a recurrent structure or a decision making construct, it does not have a defined range to signify Resolution. Most analyses performed by SRTA can proceed in the absence of the Resolution.

Considering this, the Resolution could be merged with the Constraint, or could replace the Constraint when present. It is kept as a separate range because in cases where a Resolution is present, it may disagree with the Constraint due to a defect in the validation constructs, and without a separate range for Resolution, such defects cannot be detected.

5.6.3 The Constraint

The Constraint is the range that denotes the absolute state-space for any identifier, i.e., it denotes the set of permitted values for any identifier. For example, for the most basic numeric type, it is the exact minimum and maximum values an identifier of that type can assume over a specific build system. Such ranges tend to vary across compilers and different builds of the compilers, making it necessary to incorporate build system data in the analysis. Table 5.2 details the values of the Constraint for primitive types of a C++ software under different representations.

Properties of the Constraint

The Constraint has a few significant properties.

Finite: The state-space for any entity is finite for all types in all platforms. As the Constraint signifies the absolute boundary of the state-space, it is finite too.

Continuous: The state-space is not discrete. Some states may never be achieved during a software's execution but it indeed is possible to reach these states.

Constant: For a given entity, the Constraint does not change unless the object is reallocated or redefined.

Must Exist: For the programming languages we have considered, strict type systems are enforced. No entity is possible without a type, and thus an absolute range is associated with every entity.

5.6.4 The Resolution

The Resolution signifies further restrictions put on the state-space through any validation statement. Any branching statement, loop or similar construct will contribute into shaping the state-space into a different and more restrictive one one, and will be summarized with the Resolution. In congruence to the definitions outlined in Chapter 3, the Resolution signifies the set of expected values for any entity.

Properties of the Resolution

May not be Finite: Resolution comes from the code, and is not defined by the environment. Thus it can contain defects in itself, coming from the problems in validation and loops. Wrong conditions in any loop or branching statement can cause Resolution to become an infinite range.

May not be Continuous: Resolution may not have a continuous range of values for its existence. In case of a multiple branching statement (e.g., C++ switch statement) it can have discrete values.

Variable: Resolution changes with every level of conditional statement a software's execution has to encounter, and thus can be variable.

May Not Exist: In case no conditional construct enforces any boundary on the state-space, the Resolution shall not exist.

5.6.5 The Application

The Application is the set of actual values applied on or extracted from an entity. For any entity, Application changes at every execution phase that modifies the entity.

Properties of the Application

May not be Finite: As Application signifies the values applied to the entity, it may not be finite. An infinite value is almost always an indication of a problem.

May not be Continuous: Like the Resolution, Application may consist of discrete values.

Variable: Application changes for any statement concerning a read or write of the entity.

May not Exist: Declaration of an entity does not guarantee its usage. Thus, Application may not exist for an entity. However, in case Application does not exist, it proves a problem in code.

5.6.6 Formation of the Constraint

The Constraint is formed only with the creation and destruction of entities, and remains unchanged as long as the existence of the entity is not changed.

For any single entity, the Constraint is the two extreme values the entity may assume under its build environment. For example, for a system with 2-byte integer values, a signed integer may assume values from -32768 to +32767, and an unsigned integer may assume values from 0 to 65535. The two extremes are marked by symbolic values $-\infty_{type}$ and $+\infty_{type}$. The exact values for the extremes are dependent on the system. Entries 1-4 in Table 5.4 show the situation. For pointers however, as they represent memory locations instead of variables, 0 is used as a symbolic value as the Constraint.

For entities acting as collections of other entities, like arrays (both statically and dynamically allocated), every single item in the collection has the extremes noted by its type, as described for the single entities. Additionally, the collection has an additional Constraint for its access - that is, the minimum and maximum values for indexing or access operations that specifies a valid item. For example, for an array of n elements

Table 5.4: Example Range Formations for Different Statements in C++

		Ranges			
#	Statement	C_x	R_x	A_x	Comment
1.	int x;	$[-\infty_{int}, +\infty_{int}]$			
2.	unsigned int x;	$[0, +\infty_{unsigned\ int}]$			
3.	int x = 0;	$[-\infty_{int}, +\infty_{int}]$		$[0, 0]$	
4.	const int x = value;	$[-\infty_{int}, +\infty_{int}]$		A_{value}	
5.	int *x;	$[-\infty_{int}, +\infty_{int}]$			
6.	x = new int	$[0, 0]$			
7.	x = new int[SIZE]	$[0, SIZE - 1]$			
8.	for(int i = 0; i < SIZE; i++)	$[+\infty_{int}, -\infty_{int}]$	$[0, SIZE - 1]$	$[0, SIZE - 1]$	
9.	if(x <VAL)		$[-\infty_{int}, max(A_{VAL}) - 1]$		int x;
10.	if(x <= VAL)		$[-\infty_{int}, max(A_{VAL})]$		int x;
11.	if(x != VAL)		$[[-\infty_{int}, min(A_{VAL} - 1)], [max(A_{VAL} + 1), +\infty_{int}]]$		int x;
12.	if(x < VAL1 || x > VAL2)		$[[-\infty_{int}, min(A_{VAL1} - 1)], [max(A_{VAL2} + 1), +\infty_{int}]]$		int x;
13.	x = y;			A_y	
14.	x = y+z;			$[[min(A_y) + min(A_z), max(A_y) + max(A_z)]]$	
15.	x++;			$[min(A_x), max(A_x) + 1]$	Context Dependent
16.	x;			$[min(A_x) - 1, max(A_x)]$	Context Dependent
17.	x = y;	C_y		A_y	x, y pointers to collections.

in C++ or Java, the Constraint on the array entity is $[0, n - 1]$, while for some versions of Microsoft's Visual Basic, it is $[1, n]$.

5.6.7 Formation of the Resolution

The Resolution is formed through any imposed condition on any entity that restricts the entity's state-space into a subset of the possible state-space. The restriction is enforced by checks or loop's terminal conditions.

For any conditional construct, the Resolution is formed by considering the bound(s) specified by the condition. In case a condition specifies one end, the other end is kept to the maximum reach as specified by the Constraint. If a condition, or a set of nested or sequential conditions specify both bounds, then the Resolution is formed using both bounds.

For a loop, the Resolution is inferred from the loop's initialization and condition. If the loop condition involves a complex parameter, the range, until it can be inferred correctly, is considered as the extremes.

All Resolutions must fit within the Constraint. If a situation exists where the Resolution violates the Constraint, it is a certain indication that the conditions that formed the Resolution have at least one defect.

5.6.8 Formation of the Application

The Application signifies the actual range of values an entity assumes over its course of execution. An entity assumes a value under any assignment or operation.

There are some cases, especially those that involve user-supplied values, that cannot be determined correctly without the actual program execution. In such cases the Application is approximated to the maximum possible values (i.e., worst case approximation is used).

5.6.9 The Symbolic Algebra

This section describes the symbolic algebra as it is used for the analysis. The description is organized in sections that describe particular operation and their effect on the ranges and range tuples.

Assignment

For any range to be assigned to another, regardless of the entity type they correspond to, the values in one range is copied into another. That is,

$$(R_1 \leftarrow R_2) = \begin{cases} R_1 \leftarrow [min(R_2), max(R_2)], & \text{if } ||R_2|| = 1 \\ R_1 \leftarrow \bigcup_{\forall (R_{x2} \subset R_2)} [min(R_{x2}), max(R_{x2})], & \text{otherwise} \end{cases} \tag{5.11}$$

For a statically typed programming language, when an atomic entity is assigned to another, the type or context of the receiver entity does not change, only the value is copied between entities. However, for languages like C, C++ that allows the assignment of pointers, which can be collective entities, the absolute range represented by the entities are transferred. This phenomenon is the reason to result into memory leaks. To account for this fact, for SRTA, assignments are modelled in different ways for atomic and collective entities. Therefore, for any entity y to be assigned to x, we have,

$$(T_x \leftarrow T_y) = \begin{cases} A_x \leftarrow A_y, & \text{if x is an atomic entity} \\ C_x \leftarrow C_y, A_x \leftarrow A_y & \text{if x is a collective entity} \\ T_x \leftarrow \bigcup_{\forall_z} \{T_z | \text{z is a member of y}\} & \text{if x is a composite entity} \end{cases} \tag{5.12}$$

Union

As ranges are effectively descriptions of sets, the range union operation works much the same way as set union.

$$(R_1 \cup R_2) = \begin{cases} \{R_1, R_2\}, & \text{if } ||R_2|| = 1 \\ \{R_1, \bigcup_{\forall (R_{2x} \subset R_2)} R_{2x}\}, & \text{otherwise} \end{cases} \tag{5.13}$$

Intersection

Intersection expresses the overlapping between ranges. The intersection of two ranges consist of the range that is common between them, and an empty range if no common range exists. The ranges are overlapped if the intersection does not produce an empty range.

$$(R_1 \cap R_2) = \begin{cases} [min(R_2), max(R_1)], & \text{if } R_1 \bot R_2 \text{ and } ||R_2|| = 1 \\ [min(R_1), max(R_2)], & \text{if } R_1 \top R_1 \text{ and } ||R_2|| = 1 \\ \bigcup_{\forall (R_{2x} \subset R_2)} R_{2x}, & \text{if } ||R_2|| > 1 \end{cases} \tag{5.14}$$

Arithmetic Operations

Arithmetic operations on any range applies on both extremes of the range, and only on the extremes of the range. If the range contains subranges, arithmetic operation does not have any effect on any but the terminal range.

Arithmetic operations are carried out on the Application range only. This is because, for any statically typed programming language, an arithmetic operation does not change the type of the resultant entity, although it may create a temporary entity based on the type of its operands.

$$(R_1 + R_2) = \begin{cases} [min(R_1) + min(R_2), max(R_1) + max(R_2)], & \text{if } ||R_1|| = 1 \\ \forall_{(R_{1x} \subset R_1)} [min(R_{1x}) + min(R_2), max(R_{1x}) + max(R_2)], & \text{otherwise} \end{cases} \tag{5.15}$$

$$(R_1 - R_2) = \begin{cases} [min(R_1) - min(R_2), max(R_1) - max(R_2)], & \text{if } ||R_1|| = 1 \\ \forall_{(R_{1x} \subset R_1)} [min(R_{1x}) - min(R_2), max(R_{1x}) - max(R_2)], & \text{otherwise} \end{cases} \tag{5.16}$$

$$(R_1 * R_2) = \begin{cases} [min(R_1) * min(R_2), max(R_1) * max(R_2)], & \text{if } ||R_1|| = 1 \\ \forall_{(R_{1x} \subset R_1)} [min(R_{1x}) * min(R_2), max(R_{1x}) * max(R_2)], & \text{otherwise} \end{cases} \tag{5.17}$$

$$(R_1 / R_2) = \begin{cases} [min(R_1)/min(R_2), max(R_1)/max(R_2)], & \text{if } ||R_1|| = 1 \\ \forall_{(R_{1x} \subset R_1)} [min(R_{1x})/min(R_2), max(R_{1x})/max(R_2)], & \text{otherwise} \end{cases} \tag{5.18}$$

The rules are the same for values instead of ranges.

The remainder, or modulus, operator works in a different way. As this operator returns the remainder of an entity, it does not go beyond 0 in its minimum value. The maximum value differs in relation of the two entities. If the divisor is bigger than the dividend, the maximum becomes the maximum of the divided, otherwise, it is one less than the divisor. Formally,

$$(R_1 \% R_2) = \begin{cases} [0, max(R_1)], & \text{if } max(R_1) < min(R_2) \\ [0, max(R_2) - 1], & \text{otherwise} \end{cases} \tag{5.19}$$

5.6.10 Range Inter-relation

This section describes the part of the symbolic algebra that governs the relation between ranges.

Range Containment

The term range containment, in context of SRTA, is used to denote the associativity of one range within another. A range is considered contained in another with respect to a particular extreme if that particular extreme of the contained range falls within the extremes of the container.

For any two ranges R_1 and R_2,

- R_1 is contained-over-minimum in R_2 if $min(R_{2i}) \leq min(R_1) \leq max(R_{2i})$ is satisfied for any subrange $R_{2i} \subset R_2$. This relation is expressed using $R_2 \perp R_1$ and its negation is expressed using $R_2 \not\perp R_1$.
- R_1 is contained-over-maximum in R_2 if $min(R_{2i}) \leq max(R_1) \leq max(R_{2i})$ is satisfied for any subrange $R_{2i} \subset R_2$.This relation is expressed using $R_2 \top R_1$ and its negation is expressed using $R_2 \not\top R_1$.
- R_1 is completely contained in $R2$ if $R_2 \top R1$ and $R2 \perp R1$. This relation is expressed using $B \between A$ and its negation is expressed using $B \not\between A$.
- $R_1 \top R_2$ implies $R_2 \perp R_1$ and $R_1 \perp R_2$ implies $R_2 \top R_1$.

Value Containment

The containment of a value with a range is used to express whether the value is included in the range of not. A value is contained in a range if the value falls within the bounds of any subrange the range is made of, and it is considered not-contained in a range if it does not fall into any subrange the range is made of.

That is, for any value v and a range R,

- v is contained in R if $(v \geq min(R_i) \; and \; v \leq max(R_i))$ for any $R_i \subseteq R$. Such containment is expressed using the usual set notation, $v \in R$.
- v is not contained in R if $(v \leq min(R_i) \; or \; v \geq max(R_i))$ for all $R_i \subseteq R$. Such containment is expressed using the usual set notation, $v \notin R$.

Equality

Two ranges are equal if they have exactly the same expanse. That is, if both of the extremes coincide with each other for each subrange they may contain.

$$(R_1 = R_2) = \begin{cases} max(R_1) = max(R_2), min(R_1) = min(R_2), & \text{if } ||R_2|| = 1 \\ \forall_{(R_{1x} \subset R_1)} \exists_{(R_{2x} \subset R_2)} (R_{1x} = R_{2x}), & \text{otherwise} \end{cases} \tag{5.20}$$

Adjacency

Two ranges are adjacent to each other if their opposite extremes coincide and they do not have an overlap.

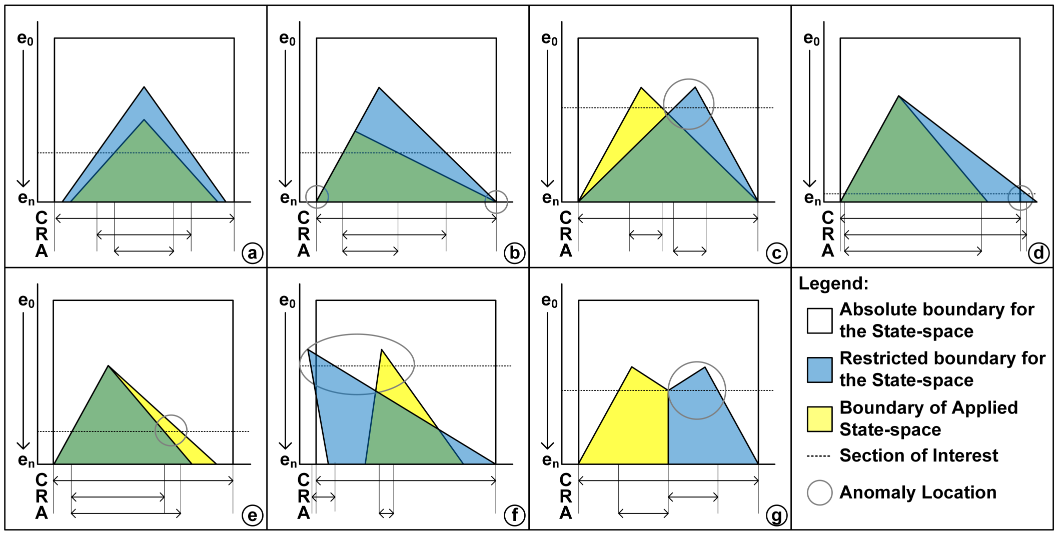

e0
en
C
R
A
a
b
c
d
e
f
g
Legend:
Absolute boundary for the State-space
Restricted boundary for the State-space
Boundary of Applied State-space
Section of Interest
Anomaly Location

Figure 5.6: Example State-space Models with Different Violations

$$(R_1 \bowtie R_2) = \begin{cases} max(R_1) = min(R_2), & \text{if } min(R_1) \leq min(R_2) \\ min(R_1) = max(R_2), & \text{if } min(R_2) \leq min(R_1) \end{cases} \tag{5.21}$$

5.7 Defect Detection from the Ranges

The model constructed with the three ranges contain the information that can be used to infer anomalies. The model is a hard-coded element, while the rules to infer defects are soft-coded, allowing the users to specify their own defect definitions. One key advantage of SRTA is to put the complexity into the hard-coded model building portion, making the specifications simple and manageable, adding to the practicality of the approach.

5.7.1 Defect Indication

At any point, any mismatch among the three ranges denotes the presence of a defect. In case no mismatch is among the ranges of a tuple, sometimes the evolution of the tuple is an indicator of a defect.

In case an Application does not agree with a Resolution, it indicates that there is a problem in either the formation of Resolution (i.e., any validation or loop construct) or in the application of the entity. The actual defect can be an integer overflow, a buffer overflow, a data mismatch or a wrong logic. In case a any other range does not agree with the Constraint, the problem is always in the range that does not agree with the Constraint.

Figure 5.6 shows examples of the defect indicators. Instead of showing the different states, the expanse of the states are shown using its boundary (triangular, as the expanse of the state-space tend to increase with execution, unless the entity is uninitialized, the Constraint is rectangular for its being a constant). Figure

Table 5.5: Defect Coverage of SRTA (with respect to the Taxonomy Developed in Chapter 3)

#	Defect	Description	Detection	Comment
1.	C1	Value Representation Defect	Full	
2.	C2	Value Offset	Full	
3.	C3	Undefined Outcome	Full	
4.	L1	Improper Checks	Full	
5.	L2	Improper Terminal Condition	Full	
6.	L3	Wrong Operation	Partial	Most cases can be detected, but some require design information.
7.	L4	Flaws in Algorithm	None	All cases require design information.
8.	L5	Performance Issues	Partial	
9.	L6	Improper Exception Handling	Full	
10.	L7	Control Flow Error	Full	
11.	L8	Design Non-conformance	None	All cases requires design information.
12.	M1	Invalid Memory Reference	Full	
13.	M2	Improper Deallocation	Full	
14.	M3	Memory Leak	Full	
15.	M4	Overflow / Underflow	Full	
16.	D1	Interface Mismatch	Partial	Some cases require architectural/design information.
17.	D2	Data Mismatch	Partial	Some cases require architectural/design information.
18.	D3	Improper Input Validation	Full	
19.	D4	Missing or Extra Input	Full	
20.	D5	Improper Abstraction	Partial	Some cases require architectural/design information.
21.	S1	Prohibitive States	None	All cases require runtime information.
22.	S2	Improper Synchronization	None	All cases require runtime information.

5.6(a) shows an ideal situation without any anomaly, as the Resolution is contained in the Constraint, and the Application is contained in the Resolution. In Figure 5.6(b), both the Resolution and the Application stay within limits, but their occupation of the entire limit of Constraint is a potential (not certain) problem. In Figure 5.6(c), the Resolution and Application do not encompass the same range, denoting a certain problem, with another instance of the same problem being displayed in Figure 5.6(g). In Figure 5.6(d), The Resolution surpasses the Constraint, indicating a certain problem, with the same occurring in a different way in 5.6(f). Figure 5.6(e) shows a situation where the Application surpasses the Resolution, denoting a certain problem.

5.7.2 Coverage

Although most defects leave traces in code, not all of them can be detected from source code alone. In the taxonomy generated in Chapter 3, all listed defects are ones that place their footprint in the source code, but some require other information like design data or architectural information to be detected. In this analysis technique, the focus is made only on defects that are contained in source code in their entirety and do not require any information from outside of source code for their detection. Table 5.5 lists SRTA's coverage of the developed taxonomy.

Among the defect classes, SRTA can detect all defects that are contained entirely within the source code (C1-3; L1, 2, 6, 7; M1-4; D3, 4). The defect L8 (Design Non-conformance) is the deviation from the design of the software and requires design information for its detection, making it out of the scope of SRTA. The two synchronization defects S1 and S2 require runtime information as the synchronization is a dynamic phenomenon. Synchronization defects are often out-of-reach for static analysis tools due to the lack of interleaving information [79] and SRTA, being a static analyzer, does not cover them.

Some defect classes include defects that do not have external information as a requirement, but often involve such information. SRTA can address the detection of these defects as long as they do not involve external information. From Table 5.5, the defect L3 (Wrong Operation) can be detected when the operation is incorrect based on the context (e.g., a loop's counter update involves an increment operation but the terminal condition requires the counter to be less than the initial value). In cases where the value requires an understanding of the underlying principles (e.g., wrong precedence), SRTA cannot detect the values correctly. However, as the L3 defects are mostly context-dependent, SRTA applies to most of them.

The defects under class L4 (Flaws in Algorithm) rely heavily on the design specifications and are mostly out of scope for SRTA. Only the very trivial defects of this type may be detected by SRTA (e.g., not updating a loop controller variable inside the loop, or updating through a branch that will never be reached). However, there is no way to conclusively state that a code fault is initiated from a flaw in the algorithm without knowing what the algorithm is. SRTA detects the defects introduced by L4 under other classes, but cannot report them as being under L4.

The defects under class L5 (Performance Issues), like the L4, mostly require program comprehension and algorithm revision to be detected, thus are out of scope for SRTA. Still, SRTA can detect the portion of such defects that stay entirely in code (e.g., checking for values that will never be used).

The defect D1 (Interface Mismatch) often involves using interfaces with wrong values but the right data types, thus are detected by SRTA. But some of the Interface Mismatch involve architectural or design specifications, and are out of scope for SRTA. For defect D2 (Data Mismatch), the same situation occurs. In case a mismatching data value is supplied by the calling context, it can be detected by SRTA, but it cannot be detected if the supplied data is wrong with respect to the algorithm, or design. The defect D5 (Improper Abstraction) is almost always a defect with explicit relation to design. SRTA can catch it only in cases where the abstraction problem is evident from code construction (e.g., a public interface returning a reference to a private property).

5.7.3 Detection Accuracy

SRTA detects all defects present in the form of anomalies by analyzing the range tuples. However, due to the nature of symbolic analysis, the range tuples may not be accurate, rather approximated. For example, if a range tuple depends on a user supplied value, there is no technique to determine the range tuple accurately without running the program. But if, after the input, a value is tested and allowed to proceed only in case it falls within the allowed range, the range can be determined accurately.

SRTA is able to detect certain and probable defects by using worst case approximations. If an user is expected to enter any value, the value is approximated to the extremes of its type - making the cases of problematic values inclusive to the range. Although this approach might increase the false positive rate, unless it is adopted, probable defects will remain undetected.

5.8 Detecting Computation Defects

SRTA can detect defects belonging to all three of the computation defect classes. The detection is performed with full coverage (i.e., any defect in any of the three classes fall within the scope of SRTA's detection capabilities).

5.8.1 C1: Value Representation Defect

A value representation defect occurs when an attempt is made to fit a value into an entity incapable of holding the value. The situation may arise from multiple causes. The value of an entity, or the outcome of an expression, can be assigned into another entity where the destination is not large enough to hold the value, or an entity's value can be modified in a way that the entity's capacity is rendered insufficient for holding it. The resultant phenomenon can be detected by checking an entity's change of Application range over subsequent statements and by checking the assignment operations for the violation of the ranges.

To detect the presence of such defects, it is required to check if the Application range of a an atomic entity is being fit into another entity for which the Constraint is not capable of accommodating the value.

To check for the anomaly, tuples formed in the following construct are checked, in statement i,

$$T_x(i) \leftarrow T_y(i) \tag{5.22}$$

For the violation of the inter-tuple relation,

$$C_x(i) \mathrel{\text{⌶}} A_y(i) \tag{5.23}$$

If a violation is detected, that is, if any bound of the $A_y(i)$ crosses those of $C_x(i)$, or if $A_y(i)$ contains a value that does not fit inside $C_x(i)$, the statement is marked as a defect.

A second indication of the defect is when an entity itself is changed beyond the range it can accommodate. In this case, the identifying anomaly is detected in any constructs in statement i, if it is detected that,

$$C_x(i) \mathrel{\not{\text{⌶}}} A_x(i) \tag{5.24}$$

where, for any previous statement, the opposite to the relation holds.

$$\exists_{(j<i)}, C_x(j) \mathrel{\text{⌶}} A_x(j) \tag{5.25}$$

In case of a value truncation or cast-down, a temporary entity shall receive the value from the symbolic expression, creating an instance of the first anomaly.

Most of the modern C, C++ and Java compilers report any type casting to lower storage types as a warning. However, SRTA differs with such detection in that, compilers do not provide any discrimination for a case being an actual problem, a potential problem, or not a problem. SRTA is able to differentiate among all three.

5.8.2 C2: Value Offset Defect

The value offset defect occurs when either or both of the limits of a range is a certain offset away from where it should be. These defects are mostly associated with conditional statements, either as normal conditional branches or inside a loop statement's header.

As these defects are defined through the conditional statements, they involve the Resolution for all instances. One simple way to infer the presence of such defects is if the disagreement between the Constraint and Resolution, or Resolution and Application is found only in one or both of the extremes.

One point to note with value offset defects is their nature to occur adjacent to the expected range of values. Hence, these defects can be characterized by the following anomalies that hold for any atomic or collective entity x,

$$A_x(i) \not\supseteq R_x(i) \; and \; A_x(i) \cap R_x(i) \neq \emptyset \tag{5.26}$$

or,

where it holds that,

$$C_x(i) \not\supseteq R_x(i) \; and \; C_x(i) \cap R_x(i) \neq \emptyset \tag{5.27}$$

If either of the conditions hold, it will identify that one of the two ranges (Constraint, Resolution or Resolution, Application) is encompassing an invalid region close to the valid regions, identifying the problem.

5.8.3 C3: Undefined Outcome

This defect occurs when the outcome of an operation cannot be predicted. Two situations may give rise to the situation, (a) if a participating entity for an operation contains an unexplained value and (b) if the entities all contain defined and determined values but their interaction is undefined.

To model scenario (a), it is difficult to model all problematic values as they change with the specific application of the entity in the software (for example, a 0 (zero) causes a problem as the denominator of an arithmetic operation, but not as a multiplier). However, the situations that correspond to such scenarios is not diverse as their outcome. Over the cases investigated for this research, the problem almost always was created by using an uninitialized value or a value that has experienced a C1 defect before its use in a statement.

Two phenomena indicate an entity's being used without initialization. The first, is having no Application range (as it occurs for collective entities) and the second is having the Application range spanning the maximum of the Constraint (as it happens for atomic entities). Therefore, a C3 defect is found for an entity if one of the following holds,

$$A_x(i) = C_x(i) \tag{5.28}$$

or,

$$A_x(i) = \emptyset \tag{5.29}$$

In case of the first anomaly, the Application range is either not set, or it is used to the entire range as the Constraint, indicating that an unchecked value is used for the Application.

Scenario (b) mainly occurs from the division-by-zero error in programming, but generalizes to other values. To detect it, the following construct needs to be addressed for atomic entities,

$$T_x(i)\ OP\ T_y(i) \tag{5.30}$$

Where the entity y can be an atomic entity or the result of an expression, OP is an operation. To indicate a defect, the following needs to hold,

$$y_i \in A_y(i) \tag{5.31}$$

where y_i is an invalid value for the identifier y. This same definition can be extended by the user for any specific value for any identifier. In case of checking for the division-by-zero, the expressions need to be tuned as, $y_i = 0$ and $OP = /$.

5.9 Detecting Logical Defects

Of the eight defect classes under logical defects, SRTA can detect any defect belonging to four of the classes (L1, L2, L6, and L7), and most of the defects belonging to two classes (L3 and L5). Due to the necessity of using architectural or design data, the other two classes (L4 and L8) do not fall within the scope of detection.

5.9.1 L1: Improper Validation

The Improper Validation defect occur for the absence of, or the mistake in, the validating constructs for any identifier before its use. The absence of validation can be detected by assessing the Resolution alone. If an entity misses a Resolution range before the first Application range is introduced, it is without a validation.

A mistake in validation is characterized by one of the two scenarios - first, if a validation does not provide a Resolution agreeable to the Constraint it is a flaw in the validation, and second, if a validation does not provide a Resolution agreeable to the Application, it may indicate a flaw in the validation.

The first scenario, or absence of validation, can be detected as, for any identifier x in statement i,

$$R_x(i) = \emptyset \ and\ A_x(i) \neq \emptyset \tag{5.32}$$

The second scenario, comparisons between Constraint, Resolution and Application is,

$$C_x(i) \not\mathrel{\mathrm{I}} R_x(i) \tag{5.33}$$

or,

$$R_x(i) \not\mathrel{\mathrm{I}} A_x(i) \tag{5.34}$$

The absence of validation anomaly is self-descriptive. The other two denote the cases where the Resolution, that is, the range formation coming from the validation is not in congruence of the other two ranges.

5.9.2 L2: Improper Terminal Conditions

The improper terminal condition, for any recurrent structure, occurs when the validation used as the terminal condition is wrong. Most of such cases will be caught as parts of C2 and L1 defects. As a generic form, this validation is indicated by the presence of any defect in the control variable for any recurrent structure. To detect it, the same procedure of C2 and L1 are applied for entities in the role of recurrence controllers.

5.9.3 L3: Wrong Operation

Wrong operations happen when the result of an operation becomes completely out-of-scope of what it should be. It is characterized by either of the two scenarios, for any entity x at statement i,

$$A_x(i) \bowtie R_x(i) \tag{5.35}$$

or,

$$R_x(i) \cap A_x(i) = \emptyset \tag{5.36}$$

The first anomaly signifies where the two ranges are adjacent, with at most one common element, while the second identifies where the two do not share any common elements.

5.9.4 L5: Performance Issues

SRTA cannot detect all performance issues as they often require design and architectural knowledge. The specific defects that SRTA can detect are,

(a) A variable was declared but never used: Detecting this defect is straightforward. For any entity that has no Resolution or Application range associated with it between its point of origination and termination is an extra and unused entity. (b) An entity was declared with wrong type. (c) Part of a collective entity was never used.

An extra variable can be detected by checking the construct,

$$\forall_{(j>i)}, A_x(j) = \emptyset \tag{5.37}$$

where i is the statement of initiation of the entity.

For Atomic entities, it can be characterized by checking the situation where, for any atomic entity x

$$C_x(i) \mathrel{\text{⌶}} A_x(i) \tag{5.38}$$

but, there exists another data type d for which,

$$C_x(i) \mathrel{\text{⌶}} C_d \tag{5.39}$$

and,

$$C_d \mathrel{\text{⌶}} A_x(i) \tag{5.40}$$

For collective entities, any disagreement between the ranges indicates an unused portion. That is, for any collective entity x, if the following holds,

$$\forall_{(j>i)}, C_x(i) \neq A_x(i) \tag{5.41}$$

it denotes a performance issue.

5.9.5 L6: Improper Exception Handling

Exception data are not straightforward entities that can be represented by the state-space. To check for the exception handling capabilities in language that support them, an intermediate representation is required to transfer the information into the symbolic domain.

A convenient workaround is to represent all exceptions thrown in a piece of code as a collective entity with indices denoting different member exceptions. Any processing of the exceptions is then modelled as an access to that collective entity with the same index translations. If a wrong exception is processed, or an exception is not processed, it is indicated by an invalid reference to that exception. That is, to formulate, if entity e holds the translated exception data, any anomaly in the formulation,

$$\forall_{(j>i)}, C_e(i) \mathrel{\not\text{⌶}} A_e(j), \tag{5.42}$$

and,

$$\forall_{(j>i)}, C_e(i) \neq A_e(j), \tag{5.43}$$

will denote a problem in exception handling. First of the two anomalies identifies a situation where at least one exception handled wrong. The second anomaly identifies that there is at least one unhandled exception in code.

5.9.6 L7: Control Flow Error

The control flow defects are characterized by situations for which the control flow is always directed towards a branch statement, or never directed to one. Detecting this defect is straightforward in the scope of any entity. For any entity x, the defect is indicated by the following constructs,

$$\forall_{(j>i)}, \cup R_x(j) \neq \cup A_x(j) \tag{5.44}$$

If this anomaly is present in code, it means that either the control flow is localized for a portion of the total space (in case $R_x(j)$ has more expanse that $A_x(j)$) or that a portion of code is never executed (if $A_x(j)$ has more expanse than $R_x(j)$)

5.10 Detecting Memory Defects

SRTA can provide full coverage to each of the four defect classes for memory defects.

5.10.1 M1: Invalid Memory Reference

An Invalid Memory Reference occurs if a memory access attempt is made for a piece of memory that does not exist. The access can be marked by a null-referrer, by an uninitialized referrer or by a referrer that holds the address to a memory already deallocated or out-of-scope.

By nature, initiation and features, this defect is similar to the C3 defect. Detecting this defect, therefore, requires similar constructs. For the detection of null pointer dereference, for any entity x representing memory, the following conditions are analyzed,

$$T_x \tag{5.45}$$

for the presence of the anomaly,

$$C_x = \emptyset \tag{5.46}$$

or,

$$C_x = [0, 0] \tag{5.47}$$

A major difference with the C3 defect is that, C3 defect applies to atomic entities, but memory entities are almost always collective. To check the entity x for an uninitialized value, it has to be checked,

$$C_x = [-\infty_{max}, +\infty_{max}] \tag{5.48}$$

which, if satisfied, indicates a defect.

5.10.2 M2: Improper Deallocation

An improper deallocation occurs when a piece of memory is deallocated that is not meant for deallocation. The same situations that give rise to an Invalid Memory Reference, may create an improper deallocation if used in a deallocation statement instead of memory access.

To check for the defect, following anomalies are checked for any memory entity, for any memory entity x deallocated at statement i

$$C_x(i) = \emptyset \tag{5.49}$$

or,

$$C_x(i) = [0, 0] \tag{5.50}$$

or,

$$C_x(i) \mathrel{\text{⌶}} [-\infty_{max}, +\infty_{max}] \tag{5.51}$$

All first three of the anomalies, identical to the anomalies for M1 defects, show the situation where an invalid memory reference is tried to be deallocated. Last of the anomalies

5.10.3 M3: Memory Leaks

Memory leaks occur when an allocated memory is not deallocated at the end of its usage. Thus, for any entity representing a reference to a memory, if the following condition exists, for any memory entity x subjected to a deallocation at statement i, or if i is the point of termination for the program,

$$\exists_{(j>i)}, C_x(j) \neq \emptyset \tag{5.52}$$

it denotes a memory leak.

5.10.4 M4: Overflows / Underflows

Memory overflows and underflows occur when a memory access is attempted beyond the permitted range of the memory. Technically, it is an invalid memory reference, done adjacent to a range of valid memory. This defect can be originated both in the validating constructs and the access statements, and hence involves the Resolution and the Application. To detect this defect, for an entity x representing a memory region, the following has to hold,

$$C_x \not\mathrel{\text{⌶}} R_x \tag{5.53}$$

where, either,

$$C_x \top R_x \tag{5.54}$$

or,

$$C_x \perp R_x \tag{5.55}$$

or,

$$C_x \bowtie R_x \tag{5.56}$$

The same applies for the Application range with the Constraint. That is, if the A_x holds the same relation with the C_x, it also indicates the same defect.

5.11 Detecting Data, Interface and I/O Defects

Among the five defect classes of this group, SRTA can provide full detection coverage to two (D3 and D4) and partial coverage to the other three (D1, D2 and D5). The partial coverage to each of the classes is able to detect most of the defects belonging to the classes.

5.11.1 D1: Interface Mismatch

In an interface mismatch, an interface is used with wrong values for data. For most cases, this defect is an instance of C1 or C3 applied to formal parameters. To detect this defect, the following constructs need to be checked. For any actual parameter y and a formal parameter x,

$$C_x(i) \not\top C_y(i) \tag{5.57}$$

This particular anomaly denotes that the type of the parameters did not match, and actual parameter is not able to hold all the values supplied to it, although it may not run into a defect at present. Other types of this defect requires the knowledge of the design for detection.

5.11.2 D2: Data Mismatch

The data mismatch defect occurs when the interface is used with proper data types, but not with proper data values. Any value supplied beyond the capability of the formal parameter is a certain indication of such defect. If the value is not beyond the capability of the formal parameter, but is never used throughout the interface, it also denotes the same defect.

Detection of this defect requires checking of the following condition, for any formal parameter x and actual parameter y

$$C_x(i) \not\top A_y(i) \tag{5.58}$$

if this does not hold, then, for all subsequent use for the entity x in the code that follows, an anomaly of

$$\exists_{(j>i)}, R_x(j) \not\top A_x(i) \tag{5.59}$$

indicates this problem.

5.11.3 D3: Improper Input Validation

The improper input validation defect contains the same characteristics as the L1 defect, only applied to the inputs of a module or the program. If the entity x is used as an input for any module or the full software itself, and it lacks validation, the situation is characterized by,

$$T_y(i) \top T_x(i) \tag{5.60}$$

where,

$$\forall_{j>i}, R_x(j) = \emptyset \tag{5.61}$$

5.11.4 D4: Missing or Extra Inputs

The missing or extra inputs defect occurs when an interface is used with insufficient or over-sufficient data. To detect the defect, following situations are analyzed,

$$\forall_{f:formal\ parameter} C_f \neq \forall_{a:actual\ parameter} C_a \tag{5.62}$$

where f is the set of all formal parameters and a is the set of all actual parameters.

For a missing input defect, in addition to the indication presented above, the following anomaly shall be in force,

$$|C_f| > |C_a| \tag{5.63}$$

or,

$$||C_f|| > ||C_a|| \tag{5.64}$$

The opposite will hold for an extra input defect

$$|C_f| < |C_a| \tag{5.65}$$

or,

$$||C_f|| < ||C_a|| \tag{5.66}$$

5.11.5 D5: Improper Abstraction

SRTA can detect improper abstractions when they are directly apparent from code. The most severe case that occurs in code is providing a reference to a private member of a class through a public function.

Under the scope of SRTA, this defect cannot be checked directly as the other ones. However, a workaround exists. If any member entity to any piece of code is reset to have the range [$\emptyset$] (empty) once out of scope. This situation then shall give rise to the other errors once accessed. Thus, the identifying signature for such defects becomes,

For any statement

$$T_y(i) \leftarrow T_x(i) \tag{5.67}$$

if,

$$C_x(i) = \emptyset \tag{5.68}$$

but,

$$\exists_{(j<i)}, C_x(j) \neq \emptyset \tag{5.69}$$

then the defect denotes an improper abstraction.

5.12 Answering the Research Questions

This section attempts to answer the research questions mentioned in Chapter 1. In particular, two research questions (RQ1 and RQ2) are answered in this chapter, with both answers reinforced with experimental evidence in the next chapter.

5.12.1 RQ1: Detecting Multiple Dissimilar Classes of Defects

The first research question for this research was stated as, "Can a specific abstraction provide sufficient means for detecting multiple dissimilar classes of defects?" The answer to this question is presented using the materials provided in this chapter. As it was presented in earlier sections, SRTA represents the entities from source code using the range-tuples. The three ranges in the tuple corresponds to three phases of an entity's life-cycle. For different classes, the rules of detection were presented in the previous section.

To illustrate the objective of the research, that is, to detect multiple dissimilar classes of defects using a single model, the rules are applied together. Each program statement passes through SRTA's detector that

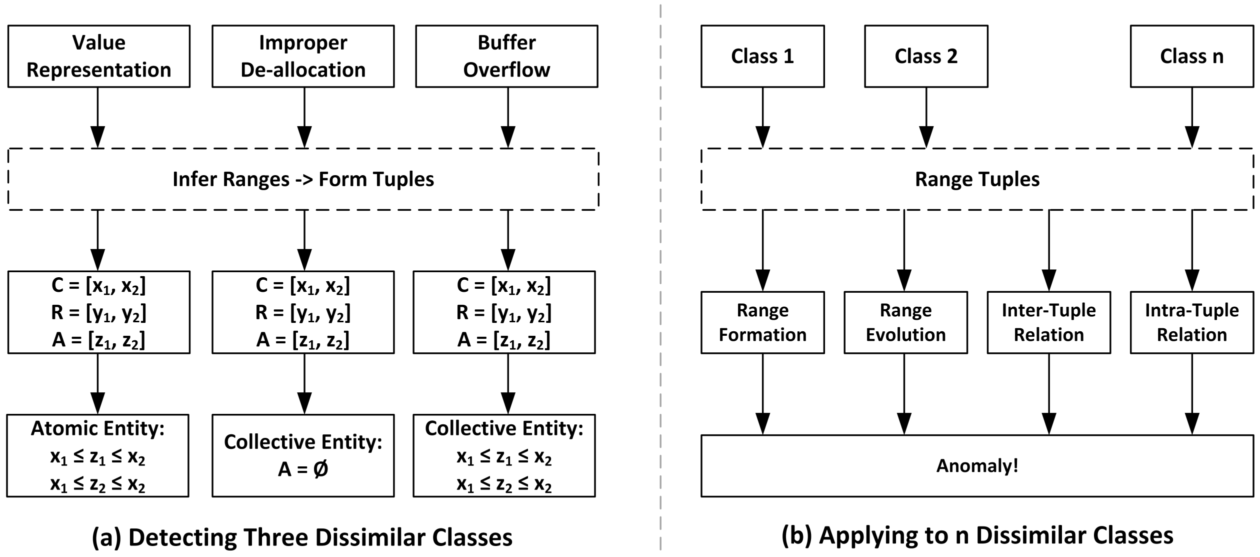


Figure 5.7: Detection Mechanism for Dissimilar Classes of Defects

checks if the statement conforms to any of the specifications. One common point that has been established in this chapter so far is that, all detection mechanisms performed on the three ranges that form the range tuple only - thus creating a single model.

Figure 5.7 illustrates the detection of dissimilar classes of defects. In Figure 5.7(a) three dissimilar defect classes are shown. Defects belonging to these three classes, once they are passed through SRTA's model builder, are expressed by using the range tuples. The range tuples have exactly the same structure for all defects - enforcing the single model methodology. The similar-in-construct range tuples coming from dissimilar defects are then checked by SRTA against specific conditions and a defect is reported once a violation is detected.

If this three-defect-case is extrapolated further, as is shown in Figure 5.7(b), we have n dissimilar defect classes, all of which are represented by SRTA using the same model (i.e., the range-tuples). The model is then checked for specifications concerning one of the four aspects, Range Formation, Range Evolution, Inter-Tuple Relation and Intra-Tuple Relation (Not all aspects are applicable to all defect classes). If an anomaly is detected in the checking, it is reported as a defect.

In summary, this process proves the ability of presenting a specific abstraction, that is, the use of range-tuples to represent and detect multiple dissimilar classes of defects.

5.12.2 RQ2: Scope of Detection

The second research question was stated as, provided a specific abstraction is possible and existent to represent multiple dissimilar classes of defects, “To what length can such a technique go in terms of dissimilar classes of defects?”.

The information presented in Figure 5.7, provides the answer to this question. As it can be seen from the figure, and as it was established in the arguments presented in earlier sections, SRTA can detect dissimilar defects by representing them in its own model of range tuples, and then by applying analyses belonging to one of the four aspects to find the anomalies. The detection process is based on the model construction. This statement can be generalized into stating that SRTA can detect the dissimilar defects as long as they are represented using SRTA's model. SRTA's model uses

In summary, this chapter hypothesizes that as long as the defects are contained in source code, they can be represented using SRTA's model and can be detected. If the defects require external data like the architecture or design, it is not possible for SRTA to represent them in a model suitable for detection.

5.13 The SRTA Prototype

To test SRTA over real-world systems, a fully functional prototype was implemented. This section describes the prototype's architecture and implementation-specific information. Due to the large volume of information related to the design and construction of the prototype, it is not described in full. Full description, along with the details on specification languages, are provided as a supplement to this dissertation [11].

5.13.1 Prototype Architecture

Figure 5.8 shows the structural view of the prototype. In the figure, the corner-folded document symbols represent external data and the standard flowchart document symbols represent internal data. Rectangles denote the structural components. Circles inside rectangles represent configurable components that depend on external specifications, and bounded rectangles represent pluggable modules as developed by users' specifications. To keep the diagram less cluttered, internal data is shown at its origination / modification point. It is to be assumed that the same internal data flows through all components that come after its origination.

5.13.2 Components

Components of the SRTA prototype are shown in Figure 5.8. The components Lexical Analyzer (LA), Token Recognizer (TR), and Token Sequence Mapper (TSM) correspond to the Preprocessing Phase. Graph Builder (GB), Range Approximator (RA), and Tuple Builder (TB) correspond to the Model Building Phase. Finally, Analyzer (AZ) and Report Generator (RG) constitute to the Analysis and Reporting Phase.

Preprocessing Phase

This phase of the prototype, corresponding to the preprocessing phase of the technique, prepares the input for processing. It includes the components for lexical analysis and token sequence mapping.

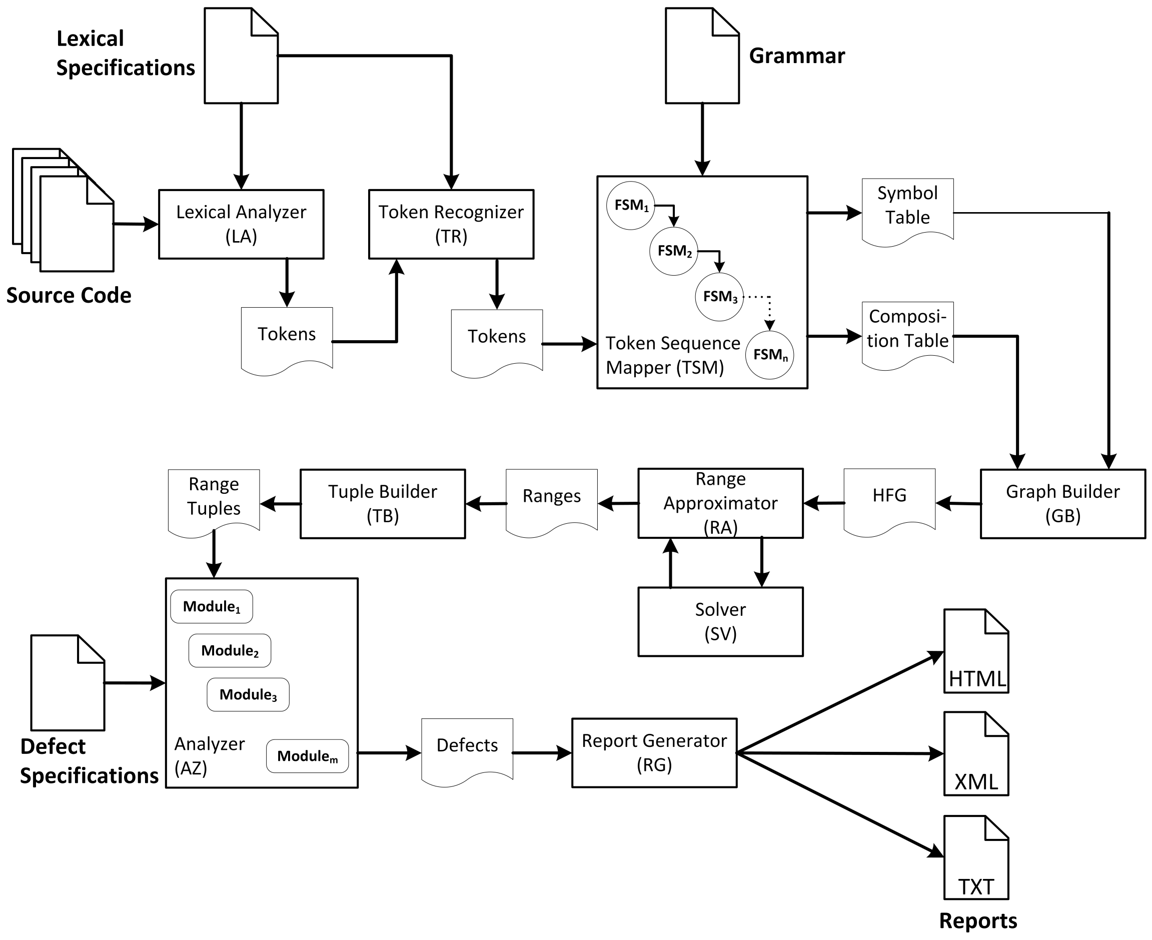


Figure 5.8: Structural Model of the SRTA Prototype

The Lexical Analyzer (LA)

Once the system reads the Source Code, the first processing unit is the Lexical Analyzer (LA) which breaks the code down into a set of tokens, based on the Language-Specific Lexical Specifications provided as external data. The token stream is then passed through a Recognizer that segregates the identifiers, using the same set of specifications.

The reason for implementing the LA, instead of using a Lexer developed by a Generator like Flex [58, 102] is that, SRTA's aim of being generally applicable is realized through its capability of processing test systems developed in different programming languages. The scanner generators develop scanners that are specific to one set of specifications only. It indeed was possible to use such scanner generators for SRTA, but it would have required source regeneration and recompiling every time a specification was changed, creating dependency on the scanner generator and on an external compiler. In the current approach, SRTA uses its own reconfigurable lexical analyzer, which does not extend beyond programming languages (unlike the scanner generators). The advantage of such a reconfigurable analyzer is that, it does not require any source modification or recompiling if the specifications are changed.

The LA passes a set of tokens, as derived from the source code, to the Token Recognizer (TR). The set contains the tokens and their location information in code. The specifications to the LA are passed as external input, which are detailed later in this chapter.

The Token Recognizer (TR)

The Token Recognizer (TR) was introduced because SRTA does not use conventional parsing, requiring a forward recognition mechanism before the tokens can be used in further processing. The application of a conventional parser could eliminate the need, but would have introduced problems of generality.

The TR accepts the set of tokens from the LA, and adds token specific information to the set, including the type of token and its location in code (File, Line, Statement). The TR passes the token sequence to the Token Sequence Mapper (TSM). The TR uses the same specifications as the Lexical Analyzer.

The Token Sequence Mapper (TSM)

After the TR processes the token set, the Token Sequence Mapper (TSM) maps the token sequences of interest. The TSM is a collection of n Finite State Machines (FSM), that identify specific patterns in the token stream as specified by the Grammar. This specific matching is similar to the LALR Parsing mechanism, but is not identical to it. Unlike conventional parsers, SRTA does not require a complete grammar. Rather, it requires only specific patterns of interest (e.g., variable declaration, initialization, assignment, increment). A conventional parser could be used for SRTA, replacing the TR and TSM components, but it would have required a complete, or at least, parsing-error-free program for analysis. In its current form, SRTA can process code that cannot be compiled due to the presence of compile-time errors, up to a point where the error does not affect the actual defective code. This specific strategy made SRTA capable of analyzing code as they are being developed, providing a strong advantage over most other static analysis tools.

The TSM builds a Symbol Table and a Composition Table. The Symbol Table, as it is used in the field of compiler design, is the table with the data of all identifiers present in the test system, along with their scope and type information. The Composition Table is unique to SRTA, it holds the detailed breakdown of composite entities with reference to the Symbol Table.

The prototype implemented the TSM as a set of seven Finite State Machines (FSM). Each FSM recognizes one token sequence from the set - declaration and initialization, composite entity creation, array/memory access, expression, assignment, allocation/deallocation and all other statements. The same token set is passed along the seven FSMs that are connected as a chain. The Symbol and Composition Tables are implemented as relations in a mysql database.

5.13.3 Model Building Phase

In the Model Building Phase, the components of the model are built using the data provided by the Preprocessing Phase.

The Graph Builder(GB)

In the Model Building Phase, the Symbol Table and Composition Table, along with the Recognized Token Sequences, are used by a Graph Builder to create a Hybrid Data and Control Flow Graph (HFG). This graph expresses the dataflow and control transfer among the entities.

Each node of the graph is the representation of an entity's state in code. In case a statement changes multiple entities, the graph contains multiple nodes for the same statement. Conditional branches are represented as forking structures in the graph, while recurrent structures are represented with loopbacks.

The Range Approximator(RA)

A Range Approximator (RA) takes over after the HFG is created, and uses the HFG, along with the Symbol Table, Composition Table and the Token Sequences to determine or approximate the range of values associated with each entity at its particular location in code. The RA uses simple mathematical relations (as described in Chapter 5) to find the values. As the values for an unbound variable are not decidable in static analysis, the RA uses a Solver Module (SV) to approximate the worst case values in such cases. The Solver Module approximates values by evaluating a series of arithmetic operations that the value had to pass through. The RA outputs a set of ranges for each identifier.

In this phase, the HFG is 'Flattened' to contain only the summarized representations. In case of forks present in the graph, all possible values of the entities involved are summarized into a single node. Recurrent structures are flattened in the same way.

The Solver (SV)

The solver accepts an expression for an entity from the RA and, consulting the HFG, Symbol Table and Composition Table, provides the range of values the entity may assume at the point of interest. In case the values cannot be determined, the solver approximates the worst possible scenario for the entities. For example, if, at any point of the program, an entity is expected to read a string from the user, its length will be approximated as infinite.

The Tuple Builder (TB)

The Tuple Builder (TB) uses this set of ranges, along with the HFG, the Symbol and Composition Tables and the token set, to infer range tuples for different entities in different points in code. TB decides the placement of ranges (operations concerning Constraint, Resolution, and Application) and processes them accordingly. At this point, SRTA's internal model building completes, and the HFG, Symbol Table, Composition Table and Tokenset are not used anymore.

5.13.4 Analysis and Reporting Phase

This phase uses multiple analyzer modules, built under users' specifications, and a reporting module that formats the output of the analyzers.

The Analyzer (AZ)

In the Analysis and Reporting Phase, the Analyzer Module (AZ) accepts the input from the Tuple Builder and checks the Range Tuples for specific patterns as specified by the Defect Specifications supplied to it. The Specifications are created to represent the defect mapping procedures described earlier in this chapter. As the specifications can be different in number and type, they can be thought of as pluggable modules for the AZ. Each of these Modules conducts exactly one specification check, although one might depend on others for its checking. The result from the AZ is a set of defects identified from the source code.

Analyzers use a certain specification for defect detection. The format of the specification, as used by SRTA, is described in brief later in this chapter.

The Report Generator (RG)

The defects are accepted in a Report Generator that formats and outputs them in one of the three supported formats (at present). The reports can be generated as Plain Text, as formatted HTML or as XML files.

5.14 User-Defined Artifacts

SRTA requires three user-defined specifications. However, none of the specifications are restricted in format by this research, as particular implementers of the tool may prefer a different representation. This section describes the contents of the specifications, not the format of the specification file.

5.14.1 Lexical Specification

The language specific lexical specifications specify the token types of the language used for the development of the test system. The objective for the specifications is to provide language specific information to the Lexical Analyzer and the Token Recognizer.

The prototype implemented the specification in three parts. First part is a set of keywords, along with a numeric type identifier for each, second part is a set of operators with associated numeric type identifiers and the third part is a set of delimiters (spaces, tabs, newlines etc.).

This specification is not a part of SRTA and may change depending on the particular implementation of the Lexical Analyzer. For example, if one decides to use a Lexer generated from Flex, it will correspond to the Flex specification.

5.14.2 Grammar Specification

The prototype uses a partial sequence mapper based on Finite State Machines, instead of a conventional parser. Grammar implementations may differ for different implementations, for this prototype, a seven-phase partial token sequence recognition mechanism was used.

5.14.3 Defect Specification

SRTA requires two information from the defect specifications. The first is the symbolic pattern to look for, and the second is the description of anomaly expected in the pattern

The patterns use a form of high-level specification language similar to the Tool Command Language (TCL). One of the main objectives of SRTA is to provide the user with the option to write simple specifications, relieving the user from complex decision making processes. The specification constructs are designed to reflect this objective. The specification has two parts. The first part contains a set of patterns represented in the symbolic analysis form, and the second part is the description of the anomalies that should be identified as defects should they occur. As an example, the C1 Defect as described earlier is specified as,

Pattern: `assign identifier1 identifier2`

Anomaly: `not {contained {application identifier2} {constraint identifier1}}`

The pattern indicates to look for the assignment of *identifier2* to *identifier1* (in the usual postfix notation typical to Tcl), and to check for the specific anomaly that the Application of *identifier2* is not contained in the constraint of *identifier1*.

The specification language contains a set of keywords and specification rules which are not presented in this chapter (due to the large volume). As the exact specifications are required to use the prototype, details are provided as a user manual to the prototype. It is available as a supplement to this dissertation [11].

5.14.4 Implementation Notes

Through its own specialized Lexical Analyzer, Token Recognizer and Token Sequence Mapper, the prototype can be extended to systems developed in any programming language that can be expressed in a context-free grammar. However, as the aim for developing this prototype was to test SRTA, and not to deliver a full-scale product, only C++ and Java Grammars were implemented for the experiments (C systems were processed using the C++ grammar).

The prototype employed mechanisms not commonly found in other tools - like the multi-phase token sequence recognition or not using a conventional lexical analyzer and parser. Also the specifications were custom designed for the prototype. The implementation and specifications are not standardized, and can be reimplemented.

5.15 Summary

This chapter details SRTA, or the Symbolic Range Tuple Analysis, technique. The chapter started by outlining the underlying propositions of SRTA, and then described the representation used by SRTA. The chapter then proceeded on describing the mathematical framework of SRTA, followed by the detection techniques for multiple dissimilar classes of defects. As the final note, the chapter described a prototype implemented to test SRTA. The next chapter Chapter 6 describes the experiments conducted to evaluate the technique.

Chapter 6
Experiments and Discussion

This chapter provides detailed information on the experiments conducted to validate and compare the technique, SRTA, as it was described in Chapter 5. The evaluations were carried out using a prototype implementation of SRTA described in the same chapter. This chapter begins with chapter specific background information (Section 6.1) and an overview of the Evaluation Activities (Section 6.2). The chapter then proceeds on describing the experimental setup for the four experiments conducted (Section 6.3). The data and analyses are presented next (Section 6.5). Finally, the chapter concludes with a summary of outcome from the experiments (Section 6.6).

6.1 Chapter-specific Background

This section provides the chapter-specific background to the evaluation of the research.

6.1.1 Accuracy

Accuracy expresses an analyzer's capability to detect as many of the relevant artifacts as possible, with the least quantity of non-relevant artifacts detected along with the relevant ones. Two popular metrics, Precision and Recall, are used to measure accuracy.

To define precision and recall, a number of background elements need to be defined. In these definition, the generic term 'Tool' is used to mean the defect analyzer being evaluated.

Let,

1. $D = \{d_1, d_2, d_3, ...d_n\}$ a collection of defective artifacts existing in the test system. The collection may contain repeated artifacts.

2. $C = \{c_1, c_2, c_3, ...c_m\}$ a set of defect classes. The set contains only unique classes.

3. $A = \{a_1, a_2, a_3, ...a_p\}$ a collection of artifacts detected by the tool as defects. The collection may contain repeated artifacts.

4. $c = M(d) : D \rightarrow C$ is the actual mapping of a defective artifact $d \in D$ to the defect class $c \in C$, known through previous detection or manual inspection.

5. $c = S(a) : A \rightarrow C$ is the mapping reported by the tool for detected defective artifact $a \in A$ to defect class $c \in C$.

6. N = Defect count. It is the total number of defects in the test system. This number is equal to the element count of the collection D. This count does not signify the unique defects existing in the system, rather counts the total number of defects, including repetition. The reason for treating repetitions as different instances is that, the same defect can be present in different flavours in different code constructs and an effective system should detect all these different flavours. If repetitions are not counted, only a generic estimate of a system's accuracy can be obtained.

7. TD = Detection Count. It is the total number of artifacts detected by the tool as defects. This number is equal to the count of elements in the collection A. The more accurate a detection system is, the closer shall TD be to N, but the opposite is not guaranteed.

8. T_P = True positive. It is the number of defective artifacts that are detected by the tool and are actually existing in the system. That is, this number is equal to the element count of the intersection of the collections D and A.

9. F_P = False positive. It is the number of artifacts detected as defects, but in reality are not defects. This number is equal to the element count for the set difference from A to D.

10. F_N = False negative. It is the number of defective artifacts actually existing in the system, but are not detected by the tool as defective artifacts.

Precision

Precision is the probability of a detected artifact's being relevant [61]. Traditionally, it is expressed as the percentage of the correctly detected artifacts with respect to all detected artifacts. To assess the tool's (specifically, SRTA's) capability in more details, two variations of precision are considered in this experiment.
Precision 1 (Lenient Estimation): In the liberal or lenient variation, the precision is considered to be the correctly detected artifacts over all artifacts. That is, a detection is considered as a true positive if the detected artifact is found to be a defect, regardless of whether the class it was reported in is the actual defect class it belongs to. This definition was used to assess the accuracy of SRTA in finding the anomalies that lead to defects, regardless of its capability of correctly interpreting them. Often the precisions reported by techniques in literature fall in this category. The expression for the lenient precision P_L, is,

$$P_L = \frac{|\{x : x \in A \ and \ x \in D\}|}{|A|} \tag{6.1}$$

Alternatively,

$$P_L = 1 - \frac{|\{x : x \in A \ and \ p \notin D\}|}{|A|} \tag{6.2}$$

Precision 2 (Strict Estimation): In the strict variation, the precision was considered to be the correctly detected artifacts under correct class over all artifacts. That is, a detection is considered as a true positive if the detected artifact is found to be a defect, and is of the same class as it was reported to be. This version of precision is used to assess SRTA's capability to correctly interpret the anomalies once they are detected. The expression for the strict precision P_S, is,

$$P_S = \frac{|\{p : p \in A \text{ and } p \in P \text{ and } M(p) = S(p)\}|}{|A|} \tag{6.3}$$

Or, alternatively,

$$P_S = 1 - \frac{|\{p : p \in A \text{ and } (p \notin P \text{ or } M(p) \neq S(p))\}|}{|A|} \tag{6.4}$$

The two variants of precision signify two different aspects of the process. Precision 1 (P_L) expresses the effectiveness of the detection of anomalies, while Precision 2 (P_S) signifies the correct interpretation of anomalies. For any test system, $P_S \leq P_L$.

Recall

Recall is the probability that a relevant item has been detected by the analyzer [61]. Traditionally, recall is expressed as the percentage of the detected artifacts over all actually present artifacts. In congruence with the same definitions for precision, we have considered two different recall measures in this experiment.

Recall 1 (Lenient Estimation): In its lenient or liberal variant, a recall is the proportion of the detected defects, regardless of the class they are detected in, to the total existing defects in the test system. That is,

$$R_L = \frac{|\{p : p \in A \text{ and } p \in D\}|}{|D|} \tag{6.5}$$

Or, alternatively,

$$R_L = 1 - \frac{|\{p : p \in D \text{ and } p \notin A\}|}{|D|} \tag{6.6}$$

Recall 2 (Strict Estimation): In the strict variant, the recall is defined as the proportion of defects detected under the correct class, to the total existing defects in the test systems.

$$R_S = \frac{|\{p : p \in A \text{ and } p \in P \text{ and } M(p) = S(p)\}|}{|D|} \tag{6.7}$$

Or, alternatively,

$$R_S = 1 - \frac{|\{p : p \in A \text{ and } (p \notin P \text{ or } M(p) \neq S(p))\}|}{|D|} \tag{6.8}$$

Like precision, the two versions of recall signify the recall in detecting anomalies and the recall in interpreting the anomalies. For any test system, $R_S \leq R_L$.

A point to note is, the words 'Strict' and 'Lenient' or 'Liberal' are used in the precision and recall specifications to indicate the relative difference in the two versions, and not to impose any general quality. The precision and recall reported by literature, which were used for comparison with other tools, all belong to the Lenient estimate.

6.1.2 Scalability

Scalability of an analyzer denotes the ability to retain its qualities against a varying range of properties of the test systems. In the context of this research, scalability is used to denote the capability of successful processing with codebases of various sizes, and with various degrees of information that can be inferred from the codebases.

A scalable technique should be able to function with codebases of varying sizes and complexities, with reasonable accuracies for each.

6.1.3 Generality

Generality is used to describe the technique's ability to be adapted in different situations. In the context of this research, these situations are considered as different input specifications, subject system diversity, and extension of the capability of detection over multiple dimensions with similar efficiency.

In the context of this research, a technique is considered to be general if it is able to process test systems of differing technologies (e.g., programming languages, frameworks, specifications) but not necessarily of different size or complexity, which are covered under scalability.

6.1.4 Practicality

Practicality is the applicability of the technique over real-world scenarios, that is, the ability to function within reasonable environmental bounds. In this research we consider practicality to be the generic and attainable requirement for the technique to function.

In the context of this research, a technique is considered to be practical if it does not put any special requirement (e.g., extra large memory, specialized hardware, special annotations) on the test system or the experimental environment, adapts itself in varying resource constraints (e.g., memory, processing power) and is able to complete its analysis in a reasonable time frame in comparison with other notable tools.

6.2 Evaluation Overview

SRTA was evaluated to assess the four aspects of its features as described in the previous section. The assessment was carried out using four different experiments (although the feature-experiment relation is not exclusive). Table 6.1 summarizes the experiment and their objectives of assessment. Details of the experiments are provided in the next section.

Table 6.1: Summary of Assessment Procedures for SRTA's Evaluation

# Experiment	Description	Accuracy	Scalability	Generality	Practicality	Relevant Sections
1. Experiment 1	(a) Application of SRTA over Six real-world test systems.	✓	✓	✓		6.4.1, 6.5.1
	(b) Application of three state-of-the-art tools over the same test systems.			✓	✓	6.4.1, 6.5.3
2. Experiment 2	(a) Application of SRTA over a Benchmark Suite.	✓				6.4.2, 6.5.2
3. Experiment 3	(a) Application of SRTA a large-scale fault-injection experiment.	✓				6.4.3, 6.5.2
	(b) Application of three state-of-the-art tools to a large-scale fault-injection experiment.				✓	6.4.3, 6.5.2, 6.5.3
4. Experiment 4	(a) Application of SRTA in a controlled experiment.		✓		✓	6.4.4, 6.5.4
	(b) Comparison of SRTA with secondary data on seven tools.			✓	✓	6.5.3

6.3 Experimental Setup

This section describes the experimental setup used to carry out the evaluation activities for SRTA.

6.3.1 Experimental Environments

All of the experiments were carried out in two different computer systems, both used for general purpose computing. No specialized hardware or software components have been used to test SRTA.

Among the two systems, the first system (System 1) was used only to measure the effect of scalability and practicality of SRTA. All of the experiments to measure accuracy and generality were conducted on the second system (System 2). The two systems' properties are summarized in Table 6.2.

Table 6.2: Description of Experimental Environments

#	System	Property	Description	Comment
1.	System 1	Processor	Intel Core i5	4 Cores
2.		Memory	2GB	
3.		Operating System	Fedora 17 64-bit	
4.		Available Secondary Memory	200GB	
5.	System 2	Processor	Intel Core i7	8 Cores
6.		Memory	4GB	
7.		Operating System	Fedora 17 64-bit	
8.		Available Secondary Memory	200GB	

6.3.2 Test Systems

The test systems were chosen from real-world open source systems that correspond to varying domains, and are implemented in three different programming languages. Table 6.3 lists their properties.

Firefox, the popular web browser, was chosen for multiple reasons. Firefox is widely used as a browser. It is a community project that accommodates different styles, specifications and standards. Moreover, Firefox has a structured and extensive bug repository that can be used to verify the detection outcomes. Same reasons were behind the selection of Thunderbird, the email client from Mozilla.

Table 6.3: Description of the Test Systems

#	System	Version	S-LOC	Language(s)	Comment
1.	Firefox	12.0	3.4M	C, C++,Java	Well known for a lot of defects
2.	Thunderbird	12.0.1	3.8M	C++, Java	Well known for a lot of defects
3.	Linux Kernel	3.1.8	9.5M	C, C++	To test scalability
4.	Sendmail	8.12.11	86K	C	To test scalability
5.	Notepad++	5.8.3	121K	C++	Stable System
6.	BlueJ	3.0.9	103K	Java	Stable System

S-LOC was reported for the portions that fall under SRTA's processing scope.
Languages specify the languages for developing the major portions (at least 1% of total LOC).

Linux Kernel was added to the test systems to assess SRTA's scalability over a large system. Linux Kernel contains inter-related modules, providing a complex and massive software system ideal for scalability verification.

Sendmail, the mailing demon from Unix, was used in a number of experiments by other researchers. It was incorporated in the test suite to be able to compare SRTA's outcome with other research described in literature.

Notepad++ and BlueJ are two code editors implemented in C++ and Java respectively. Both are well known for their stability, indirectly stating their high quality. The systems were incorporated to assess SRTA's capability of detecting tricky defects. A second reason for incorporating BlueJ was to provide means for comparison with FindBugs, one of the tools used for comparison.

6.3.3 Tools for Comparison

SRTA was compared against three tools. Each of the three tools came from Academia and is available as gratis.

UNO

UNO is a tool developed by Holzmann [68]. It is lightweight and performs local and global (i.e., intra- and inter-procedural) analysis. The tool works on C only and does not require any source annotation, specific formatting, specific structure or paradigm.

FindBugs

FindBugs, developed in the University of Maryland, is one of the most widely cited multiple defect detection tool. The tool is reported to find more than 300 defect inducing patterns in code. Despite the fact not all of these patterns are considered defective in the context of present research, and that the patterns matched by FindBugs are all from code directly, it was included to compare with SRTA due to its closest resemblance to a multiple-dissimilar defect detection tool. FindBugs works on Java Only.

SPLINT

Developed by Evans et al., SPLINT [48] has been used in a number of experiments over the years. The tool is an upgrade of the widely successful LCLint [48], and is able to detect memory defects from C code only.

6.4 Experiment Descriptions

This section provides the description of the experiments. The data are provided in the next section.

6.4.1 Experiment 1: Application on Test Systems

SRTA was applied on the six test systems as described in Table 6.3. For each of the six cases, SRTA's reports were verified using manually checking every single defect SRTA reported. For every reported defect, it was categorized in one of the three cases

Reported Under Correct Class (CC): a reported defect belongs to this class if it is a indeed a defect, and that the class reported by SRTA is the correct class the defect belongs to, according to the Taxonomy provided in Chapter 3.

Reported Under Incorrect Class (IC): a reported defect belongs to this class if it is a indeed a defect, but the class reported by SRTA is not the correct class the defect belongs to, according to the Taxonomy provided in Chapter 3.

Not a Defect (ND): a reported defect is not a defect.

The three tools, UNO, FindBugs and SPLINT were applied on two of the same test systems, Sendmail and BlueJ. UNO and SPLINT were applied on Sendmail because of their capability to process the C language only, and because Sendmail is the only system among the test systems to implemented using only C. BlueJ was processed using FindBugs, as FindBugs can process only Java Code and BlueJ was the only system among the test systems that was implemented using only Java.

6.4.2 Experiment 2: Application over Benchmark

BugBench is a benchmark developed by Lu et al. [112]. The benchmark contains a collection test systems written in C and documented defects present in them. The benchmark was used to determine the recall of SRTA. For every defect SRTA failed to detect, the reason for the failure was investigated by manual inspection of the source code. The benchmark supports L3, M2, M4 and S1 defects, as shown in Table 6.4.

Unfortunately, no benchmark could be found that encompasses the entire range of defects SRTA is able to detect. To make a comprehensive assessment of recall, Experiment 3 was used. Due to the limited defect coverage of the benchmark, it was not used to compare SRTA with the other tools.

Table 6.4: Properties of the BugBench Benchmark version 1.1

			Defect Count in BugBench Under Different Defect Classes																				
#	System	LOC	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.	bc 1.06	15k															3								
2.	cvs 1.11.4	110k													1										
3.	gzip 1.2.4	7k															1								
4.	httpd 2.0.48	319k																					1		
5.	man 1.5.h1	11k						1									1								
6.	mysql 3.23.56	508k																					1		
7.	mysql 4.1.1.α	739k																					1		
8.	ncompress 4.2.4	1.4k															1								
9.	polymorph 0.4.0	3.7k															2								
10.	Squid 2.3.	73k															1								

LOC = Lines of Code

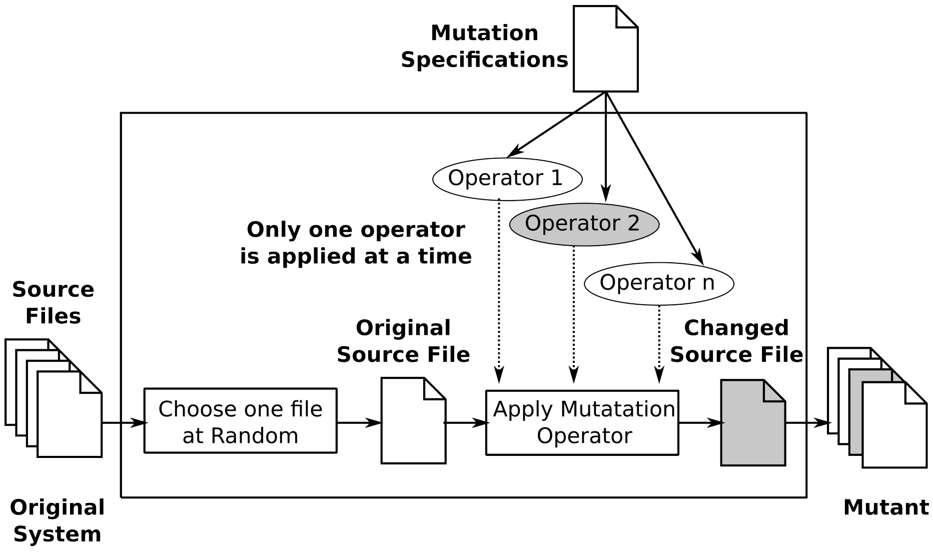


Figure 6.1: Generation Phase of the Mutation-Injection Framework

6.4.3 Experiment 3: Large-scale Fault-Injection Experiment

To evaluate the recall through the fault-injection experiment, the adaptation of an established Mutation-Injection Framework by Roy and Cordy [136, 148] for evaluating code clone detectors [137] (in particular, for evaluating the NiCad clone detector [34, 134]) was used. The framework is composed of two phases, Generation and Evaluation.

Generation phase of the framework generates the mutants (i.e., versions of the test system with induced changes), following the process as depicted in Figure 6.1 which are later used for assessment in the evaluation phase. The defect Taxonomy developed in Chapter 3 was used to develop mutation rules (i.e., Mutation Operators) by following a similar strategy of a clone taxonomy [135] that were specified using constructs of the TXL programming language. Three different sets of mutation operators were developed for C, C++ and Java, with minor difference to each other. The set of operators for C is summarized in Table 6.5. The operators for C++ and Java are provided in Appendix B. The table shows the mutation operators that apply on single statements. The operators for defects D3, D4 and D5, being interprocedural in nature,

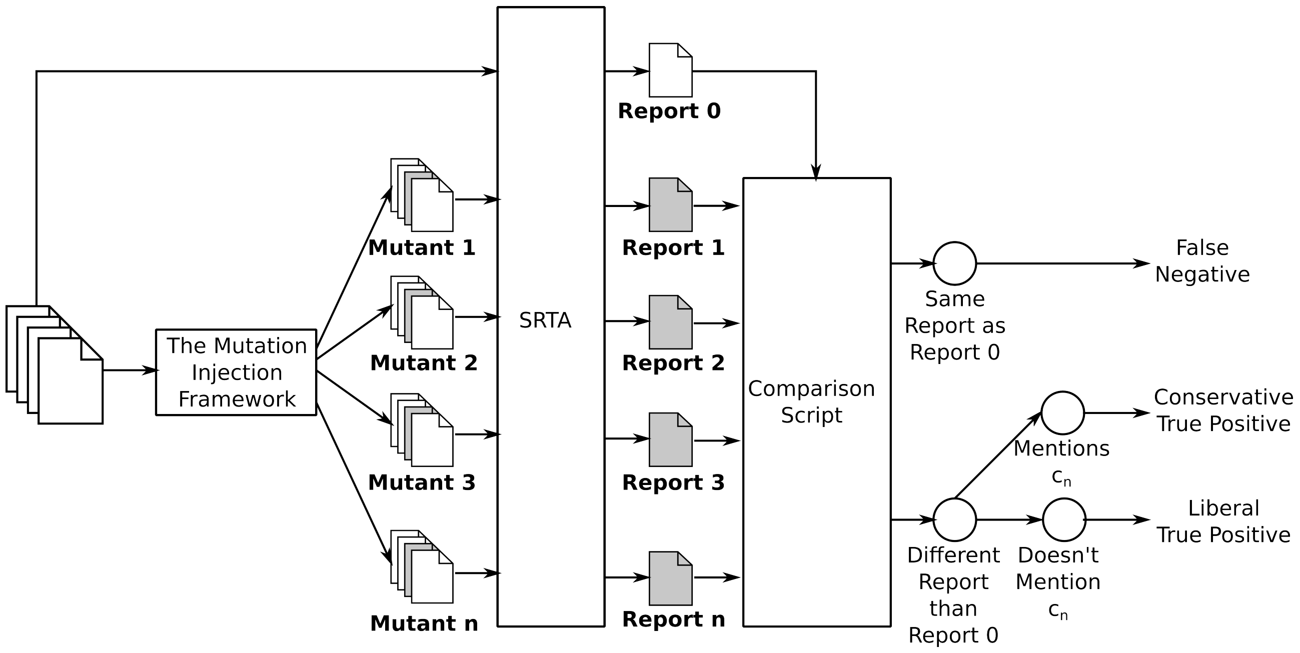


Figure 6.2: Evaluation Phase of the Mutation-Injection Framework

require complex structures and are shown separately in Appendix B.

A question may arise on the use of the operator m_{L6}, as Sendmail is a system developed in C and exception patterns are not valid in C. Among the tools evaluated, L6 pattern was supported by SRTA only. Due to the complete containment of the C constructs as valid C++ constructs, SRTA processed the C systems using C++ grammar, and thus this construct becomes valid in context with SRTA. For UNO and SPLINT, the other two tools that were applied on the same set of mutants, L6 is an out-of-scope defect class.

The un-mutated test system, shown as the 'Original System' in Figure 6.1, is mutated by choosing a random source file from the set of all files of the original system, and then by applying a mutation operator at a random position of the file. The mutation operator either changes a specific construct of the programming language to make it faulty, or introduces a new line of code to induce faults. The resultant system after the application of the mutation operator is called the Mutant. For every mutation operator, the generation process was repeated 1000 times.

In the Evaluation Phase, the mutants are processed using the analyzer under assessment and the reports are checked to determine if the analyzer was able to detect the induced defects. Due to the large number of mutants, it is often impractical to check every detection report manually. For this experiment, a specific automated procedure was developed. Figure 6.2 shows the procedure for evaluation of SRTA.

The procedure for the evaluation can be described step-by-step as,

(i) SRTA was applied on the un-mutated Sendmail. The report was marked as the Original Report, or R_0.

(ii) For every mutant $i, 1 \leq i \leq 1000$, following procedure was performed

(a) SRTA was applied on the mutant and the report was marked as R_i.

Table 6.5: Single-mutation Operators Used in this Experiment (developed for C)

#	Cls	mOP	Mutation	Example	Comment
1.	C1	m_{C1}	TP $< id >< = >< num >< ; >$	$x = 10;$	
			MP $< id >< = >< num1 >< ; >$	$x = 10 + 99999999999999;$	$num1$ = a large number
2.	C2	m_{C2}	TP $< id >< OP >< num >$	$x < 100;$	OP = $\{<, >, <=, >=, != , ==\}$
			MP $< id >< OP >< num > + < num1 >$	$x < 100 + 10;$	$num1 = [1, 10]$
3.	C3	m_{C3}	TP $< type >< id >< = >< num >< ; >$	$int\ x = 0;$	$type = int \vert float \vert double \vert long$
			MP $< type >< id >< ; >$	$int\ x;$	
4.	L1	m_{L1}	TP $< if >< (>< id >< == >< expr ><) >$	$if(id == 10)$	
			MP $< if >< (>< id >< = >< expr ><) >$	$if(id = 10)$	
5.	L2	m_{L2}	TP $< for >< (> \ldots < id >< OP1 >< num > \ldots <) >$	$for(x = 0; x < 100; x++)$	{OP1,OP2} = $\{<, >\}$,$\{>, <\}$,
			MP $< for >< (> \ldots < id >< OP2 >< num > \ldots <) >$	$for(x = 0; x > 100; x++)$	$\{<=, >=\}$,$\{>=, <=\}$,$\{<, <=\}$
6.	L3	m_{L3}	TP $< for >< (> \ldots < id >< OP1 ><) >$	$for(x = 0; x < 100; x++)$	{OP1,OP2} = $\{++, --\}$,$\{--, ++\}$
			MP $< for >< (> \ldots < id >< OP2 ><) >$	$for(x = 0; x < 100; x--)$	
7.	L5	m_{L5}	TP N/A		
			MP $< type >< id >;$	$intx;$	
8.	L6	m_{L6}	TP N/A	N/A	Inject.
			MP $< try >< \{ >< throw >< new >< type >< \} >< catch >< (>< type1 >< id ><) >$	$try\{throw\ new\ MyException\}$ $catch(Exception\ e)$	Note 1.
9.	L7	m_{L7}	TP $< if >< (> \ldots <) >$	$if(x < 100)$	
			MP $< if >< (1) >$	$if(1)$	
10.	M1	m_{M1}	TP $< id >< = >< malloc >< (>< expr ><) >< ; >$	$p = malloc(100);$	
			MP $< id >< = >< 0 >$	$p = 0;$	
11.	M2	m_{M2}	TP $< free >< (>< id ><) >< ; >$	$free(p);$	
			MP $< free >< (>< id ><) >< ; >< free >< (>< id ><) >< ; >$	$free(p); free(p);$	
12.	M3	m_{M3}	TP $< free >< (>< id ><) >< ; >$	$free(p);$	
			MP $< / >< / >< free >< (>< id ><) >< ; >$	$//free(p);$	
13.	M4	m_{M4}	TP $< id >< [>< num ><] >$	$p[30];$	
			MP $< id >< [>< num >< + >< num1 ><] >$	p[30+10];	
14.	D1	m_{D1}	TP $< id >< (>< id_1 >< , >< id_2 > \ldots < , >< id_n ><) >$	$function(x, y, z);$	
			MP $< id >< (>< id_1 >< , >< id_1 > \ldots < , >< id_1 ><) >$	$function(x, x, x);$	
15.	D2	m_{D2}	TP $< id >< (> \ldots < num > \ldots <) >$	$function(10, 20, 30);$	
			MP $< id >< (> \ldots < num1 > \ldots <) >$	$function(10, 0, 30);$	
16.	D3	m_{D3}	Complex Operator. Shown in Appendix B		
17.	D4	m_{D4}	Complex Operator. Shown in Appendix B		
18.	D5	m_{D5}	Complex Operator. Shown in Appendix B		

Cls = Defect Class
TP = Token Pattern
<> = Token
mOP = Mutation Operator
MP = Mutation Pattern

Note 1: Although it is not recognized for C, as SRTA processed the C Systems using C++ grammar, it counts as a valid construct.

(b) If R_i is the same as R_0, SRTA is considered to have missed the defect.

(c) If R_i is the not the same as R_0, but SRTA's report does not mention the defect class in the mismatching portion, SRTA is considered to have detected the defect in a different category than it should be reported into.

(d) Otherwise, SRTA is considered to have detected the defect correctly.

Same set of mutants were used with SPLINT and UNO to assess their recall to compare with SRTA. For FindBugs, another set of mutants were generated using BlueJ as the test system, and both SRTA and FindBugs were applied on the same set of mutants for the comparison of results.

To compare the reports R_0 and R_i, the plain `diff` utility was used for SRTA. The utility reports a line-by-line match or mismatch on the source files compared. In case the reports are the same, it is taken as an indication of SRTA's failure to detect the defect. In case `diff` reported a mismatch, the mismatching report fragment from R_i is checked for the mention of the defect class the mutant was developed for. If the class is found in this portion, SRTA is considered to have detected the defect correctly, if not, then SRTA is considered to have detected the defect in an incorrect class.

SRTA provides an added advantage over the other three tools in that, SRTA does not involve a conventional compiler like gcc to build the code. This feature provides SRTA with the capability to process incomplete code, and relieves it from another problem present in the other systems. In case of multi-file compilation, the exact compilation order of the files are not usually decidable for gcc. This problem can cause UNO and SPLINT, when applied on the same subject system more than once, to generate reports that are same in content, but different in ordering. Although the problem was not observed for FindBugs, no information was found to guarantee its absence.

As a workaround to this problem, the tools were used to generated plain text reports, which were then converted to XML using a custom script. The XML report contents were then compared using an evaluation version of the XML DiffDog by Altova [3].

6.4.4 Experiment 4: Controlled Experiment

SRTA was implemented to balance between available memories. SRTA uses the available primary memory as much as possible. If it runs out of memory, it moves the data to secondary memory and loads them as required. The data unit granularities can be as small as the set of tokens from a single file. By design, SRTA is guaranteed to run as long as the primary and secondary memories of the computer system it is executed on do not run out together, contrary to most other tools that only rely on the primary memory.

To test the scalability of SRTA, two different execution systems, as described earlier in this chapter, were used. In each of the systems, five small subject systems were processed. As these systems were only used to check scalability and practicality, and as SRTA's processing accuracy does not depend on the size of the subject system (as was apparent from the precision experiment), the accuracy for these systems were not verified. Default Linux profiling tools `time` and `vmstat` were used to log the execution information.

As the target of this exercise was only to verify SRTA's scalability and resource usage, instead of an independent software, five collections of SRTA's own source code (including repetition) were used. To estimate the least requirement for the software, the smallest possible C++ code was used (that is, the one that contains a main function and a return statement inside). From SRTA's Source Code, a collection was formed with 13 source files, containing a total of 3402 source lines excluding comments. Multiple copies of these files were used to form five small collections that were used to evaluate SRTA's processing time and memory over two execution systems. The size for the test systems are shown in Table 6.6.

The reason to use SRTA's own code in repetitive collections, instead of a set of established software is to have better control over the number of tokens and the codebase sizes. Instead of the entire code of SRTA, the 13 files that constitute to the common components utilized by SRTA were used because the code in these files, being generic in nature, provides greater complexity than the rest of the system.

Table 6.6: System Metrics for Scalability Test

#	Name	Files	LOC	Tokens	Comment
1.	Base	1	3	10	Baseline. Almost no processing.
2.	TS1	13	3,402	12,062	
3.	TS2	26	6,804	24,124	2xTS1
4.	TS3	65	17,010	60,310	5xTS1
5.	TS4	130	34,020	1,200,620	10xTS1
6.	TS5	1300	3,40,200	1,20,06,200	100xTS1

6.5 Results and Analysis

This section provide the results and analysis obtained from the four experiment as described in the previous section. In presenting the data, experiment-wise grouping was not used due to the same aspect being verified in multiple experiments.

6.5.1 Precision

Table 6.7 shows the results from Experiment 1. As it is apparent from Table 6.7, SRTA performs particularly well for memory and computation defects, but not so well for logical and data related defects. The reason for the performance loss in Logical and Data related defects is their dependence on artifacts external to code. Computation and Memory defects can be completely traced from code, a task SRTA is specifically designed to do. Analysis of the false positives yielded three major reasons behind SRTA's failure to detect them.

Worst-case Approximations: Worst-case approximations resulted in a number of false positives in the processing. In C1 defects, SRTA produced two false positives for each of Firefox, Thunderbird and Linux Kernel. All six of the false positives included user supplied values, and thus were approximated using the worst-case estimation which included infeasible states. Same reason was behind one false positive in C2, all

Table 6.7: Precision of SRTA in Processing the Test Systems

		Test Systems																		
		Firefox			Thunderbird			Notepad++			Sendmail			Linux			BlueJ			
#	Class	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	Comment
1.	C1	0.67	0.83	0.17	0.73	0.87	0.13	1.00	1.00	0.00	0.75	1.00	0.00	0.75	0.93	0.07	1.00	1.00	0.00	
2.	C2	0.70	0.80	0.20	0.76	0.82	0.18	-	-	-	0.50	0.50	0.50	0.86	0.93	0.07	1.00	1.00	0.00	
3.	C3	1.00	1.00	0.00	0.77	0.82	0.18	0.50	1.00	0.00	0.83	0.83	0.17	0.74	0.84	0.16	0.67	1.00	0.00	
4.	L1	0.33	0.33	0.67	0.40	0.60	0.40	-	-	-	0.50	0.75	0.25	0.64	0.79	0.21	0.56	0.67	0.33	
5.	L2	0.25	0.50	0.50	-	-	-	-	-	-	-	-	-	0.70	0.80	0.20	1.00	1.00	0.00	
6.	L3	-	-	-	-	-	-	-	-	-	-	-	-	0.67	0.83	0.17	1.00	1.00	0.00	
7.	L5	-	-	-	0.60	0.60	0.40	0.67	0.67	0.33	0.67	0.67	0.33	0.76	0.76	0.24	0.50	1.00	0.00	
8.	L6	-	-	-	-	-	-	-	-	-	-	-	-	0.75	0.75	0.25	1.00	1.00	0.00	
9.	L7	0.67	0.67	0.33	1.00	1.00	0.00	1.00	1.00	0.00	-	-	-	-	-	-	0.50	0.75	0.25	
10.	M1	0.71	0.82	0.18	0.33	0.67	0.33	0.50	1.00	0.00	0.40	0.60	0.40	0.73	0.82	0.18	0.86	0.86	0.14	
11.	M2	0.67	0.67	0.33	1.00	1.00	0.00	-	-	-	-	-	-	0.75	0.75	0.25	-	-	-	
12.	M3	0.67	0.83	0.17	0.67	0.67	0.33	1.00	1.00	0.00	0.50	1.00	0.00	0.74	0.78	0.22	-	-	-	
13.	M4	0.75	0.75	0.25	0.75	0.75	0.25	-	-	-	0.60	0.80	0.20	0.64	0.71	0.29	0.67	0.83	0.17	
14.	D1	-	-	-	-	-	-	-	-	-	0.50	1.00	0.00	0.43	0.57	0.43	0.67	1.00	0.00	
15.	D2	-	-	-	-	-	-	0.00	0.00	1.00	0.00	0.00	1.00	0.54	0.92	0.08	0.50	0.50	0.50	
16.	D3	0.70	0.85	0.15	-	-	-	1.00	1.00	0.00	0.50	0.50	0.50	0.82	0.91	0.09	0.75	1.00	0.00	
17.	D4	0.50	0.50	0.50	-	-	-	-	-	-	-	-	-	0.80	0.80	0.20	-	-	-	
18.	D5	-	-	-	-	-	-	1.00	1.00	0.00	-	-	-	1.00	1.00	0.00	-	-	-	

P_S = Precision (Strict Estimation)
P_L = Precision (Lenient Estimation)
F_P = False Positive Rate

in L1, all in L2, six in M1, twenty in M3, four in M4, and most in D1-5. Incorporation of simple heuristics in the solver and in the range tuple inference module can act against this problem.

Insufficient Library Function Models: Three of the false positives in C2 defects were due to function-dependent conditions used in checks, for which SRTA did not have the model of the functions (as they were standard library functions, source code was not provided), and therefore created the worst-case approximations. Same reason created one false positive in L2, eight in L5, one in L7, ten in M1, and seven in M3. This problem can be resolved if user-defined models of library functions can be provided to SRTA. As SRTA only deals with range tuples, it is possible to provide library function models by specifying the input and output ranges. However, this functionality was not implemented for the prototype.

Dataflow Problems: SRTA failed to properly process the cases where one entity is reused multiple times. In case a variable is declared as a loop index, and used in more than one loops in the same block of code, SRTA marked it as a defect. This situation occurred for one false positive in L2, two in L3, seven in M1, two in M2, three in M3, three in M4, two in D1, one in D2, one in D3 and one in D4. Resolution of this problem requires remodelling of dataflow, which we leave as a future research direction.

Comparing with UNO, FindBugs and SPLINT, the precisions are reported in Table 6.8.

As it can be seen from Table 6.7 and Table 6.8, SRTA outperforms UNO in all directions. The reason behind it is the same as it was reported by previous research [168, 141]. UNO only can process straightforward expressions that do not require any inference. SRTA, on the other hand, can process inferred information, and can consequently, can detect more defects than UNO.

FindBugs is a generally well performing tool that matches bug patterns and is reportedly able to find 300 "patterns" [8]. In case of L1 and L5 defects, FindBugs outperformed SRTA, although the performance was

Table 6.8: Precision of UNO, FindBugs and SPLINT Using Direct Experimental Data

#	Tool	Language	Class	Sendmail P_S	Sendmail P_L	Sendmail F_P	BlueJ P_S	BlueJ P_L	BlueJ F_P	Comment
1.	UNO	C	C3	0.75	0.75	0.25				
			M1	0.00	0.00	1.00				
			M4	0.25	0.25	0.75				
2.	FindBugs	Java	C1				1.00	1.00	0.00	
			C2				1.00	1.00	0.00	
			C3				0.67	0.67	0.33	
			L1				0.71	0.71	0.29	
			L3				0.67	0.67	0.33	
			L5				1.00	1.00	0.00	
			L7				0.67	0.67	0.33	
			M1				0.80	0.80	0.20	
			D1				0.50	0.50	0.50	
			D2				0.33	0.33	0.67	
			D3				0.75	0.75	0.25	
3.	SPLINT	C	M1	0.50	0.50	0.50				
			M2	-	-	-				None was present
			M3	0.50	0.50	0.50				
			M4	0.38	0.38	0.62				

P_S = Precision (Strict Estimation)
P_L = Precision (Lenient Estimation)
F_P = False Positive Rate

close in other defects. The reason can be attributed to FindBug's specialized treatment of Java, which SRTA had to give up to become a general tool. Many of the patterns FindBug reported (e.g., dead parameter) are specific to Java, for which SRTA does not provide specialized treatments.

SPLINT's performance over Sendmail was not above SRTA. For all categories, SRTA performed well above SPLINT. The reason for this is SRTA's incorporation of defect information from multiple phases of the entity's life-cycle, which helped to reduce false positive and false negative. This reason was found by inspecting two of the defects that SRTA detected, but SPLINT missed. For both cases, it was found that local information did not suffice in identifying the defects, but compared to their immediate conditional block (i.e., if statement), the defect became apparent.

6.5.2 Recall

The data from SRTA's application on the benchmark is presented in Table 6.9. Detailed detection counts are presented in Appendix C.

Table 6.9: SRTA's Recall Assessment by Application on BugBench

# Class	bc R_C	bc R_L	bc F_N	cvs R_C	cvs R_L	cvs F_N	gzip R_C	gzip R_L	gzip F_N	man R_C	man R_L	man F_N	ncompress R_C	ncompress R_L	ncompress F_N	polymorph R_C	polymorph R_L	polymorph F_N	squid R_C	squid R_L	squid F_N	Comment
1. L3	-	-	-	-	-	-	-	-	-	1.00	1.00	0.00	-	-	-	-	-	-	-	-	-	
2. M2	-	-	-	1.00	1.00	0.00	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
3. M4	0.75	1.00	0.00	-	-	-	1.00	1.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00	0.50	1.00	0.00	1.00	1.00	0.00	
Avg	0.75	1.00	0.00	1.00	1.00	0.00	0.50	1.00	0.00	0.50	0.50	0.50	1.00	1.00	0.00	0.50	1.00	0.00	1.00	1.00	0.00	

R_S = Recall (Strict Estimation)
R_L = Recall (Lenient Estimation)
F_N = False Negative Rate

As can be seen from the table, SRTA provided high recall in all cases except the one defect in `man`. The defect was caused by having a buffer overflow in line 979 of the file man.c, where the estimate of a loop exit condition was set 4 times higher than it should have been in the expression,

"$if(end == NULL || i + 1 == sizeof(tmp\_section\_list))$"

, which, in a correct version, should have been,

"$if(end == NULL || i + 1 = sizeof(tmp\_section\_list)/sizeof(char*))$"

, as specified in BugBench's descriptions. However, in SRTA's symbolic execution, the first condition evaluated to false only, minimizing the need to evaluate the second condition. This lazy evaluation technique, introduced to increase the prototype's performance, was the underlying reason for missing this defect.

The data from the fault-injection experiment is presented in Table 6.10.

Table 6.10: SRTA's Recall Assessment from the Mutation-Injection Experiment

				Sendmail			BlueJ			
#	Defect	mOP	Mutants	R_S	R_L	F_N	R_S	R_L	F_N	Comment
1.	C1	m_{C1}	1000	0.89	0.89	0.11	0.89	0.90	0.10	
2.	C2	m_{C2}	1000	0.86	0.91	0.09	0.86	0.87	0.13	
3.	C3	m_{C3}	1000	0.88	0.88	0.12	0.78	0.79	0.21	
4.	L1	m_{L1}	1000	0.62	0.73	0.27	0.63	0.64	0.36	
5.	L2	m_{L2}	1000	0.77	0.88	0.12	0.81	0.85	0.15	
6.	L3	m_{L3}	1000	0.71	0.79	0.21	0.73	0.75	0.25	
7.	L5	m_{L5}	1000	0.73	0.85	0.15	0.69	0.71	0.29	
8.	L6	m_{L6}	1000	0.67	0.85	0.15	0.75	0.77	0.23	
9.	L7	m_{L7}	1000	0.79	0.96	0.04	0.79	0.80	0.20	
10.	M1	m_{M1}	1000	0.82	0.88	0.12	0.90	0.90	0.10	
11.	M2	m_{M2}	1000	0.79	0.88	0.12	-	-	-	Not Defined for Java.
12.	M3	m_{M3}	1000	0.84	0.92	0.08	-	-	-	Not Defined for Java.
13.	M4	m_{M4}	1000	0.82	0.90	0.10	0.89	0.93	0.07	
14.	D1	m_{D1}	1000	0.71	0.81	0.19	0.73	0.74	0.26	
15.	D2	m_{D2}	1000	0.75	0.85	0.15	0.67	0.68	0.32	
16.	D3	m_{D3}	1000	0.62	0.78	0.22	0.60	0.70	0.30	
17.	D4	m_{D4}	1000	0.70	0.82	0.18	0.75	0.91	0.09	
18.	D5	m_{D5}	1000	0.69	0.83	0.17	0.59	0.70	0.30	
	Average		1000	0.76	0.86	0.14	0.78	0.81	0.19	

mOP = Mutation Operator
R_S = Recall (Strict Estimation)
R_L = Recall (Lenient Estimation)
F_N = False Negative Rate

On an average, SRTA detected 86% of the defects injected in Sendmail, and correctly interpreted 76% included in those 86%. For BlueJ, the detection performance is 81%, with correct interpretation of 78%. This recall performance is considerably high for a unified detection tool and is better than some of the specialized tools as well (as described in later sections).

As a general trend, SRTA showed a better recall performance in the Memory Defects (M1-M4) and on the Computation Defects (C1-C3). For both, the detection accuracies are higher, with lower misclassification problems. The reason can be traced to the structure of these defects that make the anomaly apparent. All of the computation defects are intra-procedural, and therefore do not require complex reasoning over procedural boundaries. The memory defects are sometimes inter-procedural, but are always local to the place of occurrence. Logical Defects, on the other hand, require inter-procedural analysis, often relying heavily on the context and specific execution path. In most cases, Logical Defects require a combination of the context and path for their determination, giving rise to confusion in the analysis.

For the defects under the class C1, the mutation rule was a certain one to introduce a defect. For Sendmail, SRTA did not mis-classify any of the defects from this category, but failed to capture 11% of the total injected values. For BlueJ, SRTA mis-classified 1% of injected defects, and missed 10%. Checking ten random missed defects confirmed all of them to be in function declarations in Sendmail, where the assignment of the value only becomes a default value for the function parameter. If the function is not called with this value, the context will override the defect into a safe statement, which happened for all the checked cases for these defects. The mis-classified defects in BlueJ were all classified as Invalid Memory References (M1). Manual verification of the 12 missed defects found all mutations to be implemented in variables that were used as array indices, thus actually causing the Invalid Memory References.

For the defects under the class C2, SRTA was able to find 91% of the defects in Sendmail, and correctly identified 86% of the total defects with the rest 9% as false negatives. For Bluej the detection performance was 87%, identification 86% and false negatives 13%. However, these false negative rates are not guaranteed as the mutation rule may introduce mutations that do not result into a defect. Ten random samples of the missed defect instances were checked, of which three were found to be non-defects (two in BlueJ and one in Sendmail). For the other seven, SRTA missed the defects due to the extra complex nature of the loop control structures that contained the defects. The structures involved composite entities and / or values from inside the loop's body that skipped detection.

For the defects under the class C3, like those under the class C1, SRTA did not mis-classify any defect for Sendmail, detected 88% and missed 12%. For BlueJ detection performance was 79%, with 78% identified correctly and 21% false negatives. However, the mutation rule was not certain to introduce a defect. Therefore, the missed 12% and 21% can not be considered as the actual false negative rate. Ten random defects were checked from among the missed defect instances, and it was found that for all instances, although the entities were initialized (which were changed to 'uninitialized' by the mutation operator), the uninitialized entities were not used in any of the computations, rendering the defective mutations into safe ones.

For the defects under the class L1, SRTA had a relatively high false negative rate of 27% for Sendmail and 36% in BlueJ. However, this defect class is another where the mutation operator is guaranteed to produce a defect. A random check of ten missed defects verified that the mutation operator changed the operators in complex arithmetic expressions involving true/false values, which were set to tautologies and thus skipped detection.

For the defects of the class L2, SRTA had a false negative rate of 12% for Sendmail and 15% for BlueJ. However, the mutation operator was not certain to inject a fault in this case. The mutation operator for this particular defect changes an operator with another, creating an improper validation statement. Of the ten randomly selected defect instances from among the ones SRTA failed to detect, two were not defects despite the change in operators. For the others, the operators were replaced in complex comparison operations that did not register as anomalies for SRTA.

For the defects of the class L3, SRTA's false negative rate was 21% for Sendmail and 25% for BlueJ, but the mutation operator was not guaranteed to insert a fault. Of the ten randomly chosen defects from the ones SRTA failed to detect, four were not defects, for the others, SRTA failed to detect the defects due to the ambiguity used in the precedence of the operators.

For the defects of the class L5, SRTA had a 15% false negative rate for Sendmail and 29% in BlueJ. Of the ten randomly chosen defects checked from here, eight were found to be defined later in the program and therefore were not defects. The other two SRTA missed because of their insertion in wrong places. Both of these two cases were errors to stop building the system. SRTA does not involve a compiler to be able to parse incomplete code, and thus was not able to find these cases.

For the defects of the class L6, SRTA's false negative rate was 15% for Sendmail and 23% for BlueJ. Verification with the 10 randomly sampled defects pointed out the cause for missing the defects to be the possibility of automatic conversion of the exception classes which rendered them improper for classification as defects. This particular incident was more prominent in BlueJ.

For the defects of the class L7, false negative rate was 4% only for Sendmail, but 20% for BlueJ. Of the 40 defects that SRTA failed to classify for Sendmail were due to the involvement of outer-nested conditions that rendered the inside ones improper. In case of BlueJ, ten randomly verified defects identified three of them to be from the same loop nesting conditions, and the rest due to the use of user-specified variables.

For the defects belonging to the M1 class, false negative rate was 12% for Sendmail and 10% for BlueJ. An analysis of the random sampled 10 defects revealed that for four of them, there was no defect as the values, although were initialized at the beginning which the mutant changed, was re-initialized through a validation statement and thus nullified the defect.

For the defects belonging to the M2 class, with a false negative rate of 12% for Sendmail, SRTA failed to detect all of the defects that were checked, primarily due to the multiple branching deallocation statements. This particular defect class is not valid for Java, which involves automatic garbage collection, and therefore this particular set of mutation was not performed for BlueJ.

For the defects under the class M3, with a false negative rate of 8%, all of the randomly checked ten missed defects were actually defects. SRTA missed the trivial memory leaks due to their occurrence in very complex code structures (one with a four-level nesting). This defect class, also, is not valid for Java, and was omitted for BlueJ.

For the defects under the class M4, with a false negative rate of 10% for Sendmail and 7% for BlueJ, none of the ten randomly checked defects were actually defects. The buffer size for these cases were large enough to accommodate the increased access induced by the mutant.

For the defects under the class D1, the false negative rate was 19% for Sendmail and 26% for BlueJ. Of the ten randomly checked defects it was found that nine of the defects were not actually defects as the data provided to the functions were valid. The last one had an implicit type casting, which SRTA missed.

For the defects of the class D2, none of the ten random sampled missed defects were found to be defects.

The mutant was not guaranteed to include a defect in this case.

For the defects of the class D3, with a false negative rate of 22% for Sendmail and 30% for BlueJ, the ten randomly checked missed defects were due to three different causes. First, three of the mutations were reverted by the later code constructs. Second, two of the mutations did not induce defects because the calling context of the function validated the values and the rest were due to not using all the function parameters.

For the defects belonging to class D4, with an 18% false negative for Sendmail and 9% for BlueJ, the ten randomly verified missed defects were found to be actual defects, which SRTA missed due to the composite entities involved.

The the defects belonging to the class D5, with 17% false negative rate for Sendmail and a high 30% for BlueJ, the ten randomly verified defects were found to be actual defects. For BlueJ, the defects avoided detection due to the static declaration context, which made their values reset right after the mutation.

The same experiment, repeated on the three other tools, resulted in the data provided in Table 6.11

Table 6.11: Recall of UNO, FindBugs and SPLINT Using the Mutation-Injection Experiment

							Sendmail			BlueJ			
# Tool	System	Lang.	LOC	Class	mOP	Count	R_S	R_L	F_N	R_S	R_L	F_N	Comment
1. UNO	Sendmail-8.12.11	C	102k	C3	m_{C3}	1000	0.66	0.66	0.34				
				M1	m_{M1}	1000	0.43	0.43	0.37				
				M4	m_{M4}	1000	0.00	0.00	1.00				
2. FindBugs	BlueJ-3.0.9	Java	86k	C1	m_{C1}	1000				0.90	0.90	0.10	
				C2	m_{C2}	1000				0.87	0.87	0.13	
				C3	m_{C3}	1000				0.79	0.79	0.21	
				L1	m_{L1}	1000				0.68	0.68	0.32	
				L3	m_{L3}	1000				0.70	0.70	0.30	
				L5	m_{L5}	1000				0.70	0.70	0.30	
				L7	m_{L7}	1000				0.68	0.68	0.32	
				M1	m_{M1}	1000				0.88	0.88	0.12	
				D1	m_{D1}	1000				0.65	0.65	0.35	
				D2	m_{D2}	1000				0.79	0.79	0.21	
				D3	m_{D3}	1000				0.77	0.77	0.23	
3. SPLINT	Sendmail-8.12.11	C	102k	M1	m_{M1}	1000	0.61	0.61	0.39				
				M2	m_{M2}	1000	0.57	0.57	0.43				
				M3	m_{M3}	1000	0.60	0.60	0.40				
				M4	m_{M4}	1000	0.51	0.51	0.49				

R_S = Recall (Strict Estimation)
R_L = Recall (Lenient Estimation)
F_N = False Negative Rate

As it can be seen from the table, despite missing a number of defects in the experiment, SRTA's performance is not behind the other three tools in any of the cases. SRTA outperformed UNO and SPLINT in almost all cases, and performed either above or equal to findbugs in most cases.

6.5.3 Comparison with Other Tools

This section describes SRTA's comparison with other tools. The evaluation was carried out in two phases. First, SRTA was compared against the three tools mentioned earlier using experimental data. Second, the data obtained from the experiments were compared against the data reported in literature for seven tools, five of which were not used in the direct comparison.

Comparison Using Data Obtained from Experiment

The results of the direct comparison, using all defect classes described in the Taxonomy in Chapter 3, are presented in Table 6.12. In the table, as SRTA was involved in two Mutation-Injection experiments, the data was used from two different tables. If, under a defect class, a direct comparison was made with other tools, the average of the direct comparison data was used for SRTA. In absence of such comparison, the data from Table 6.7 and Table 6.10 was used. For the other three tools, the precision and recall were taken from Tables 6.8 and 6.11.

In Table 6.12, it can be observed that SRTA clearly covers more defect classes than the other three tools. Compared to UNO's support to three classes under two groups, FindBug's 11 defect classes under three groups, and SPLINT's four under one group, SRTA supports 18 defect classes in three groups, with individual precision and recall as good as, if not better, than the other three tools in different classes. Only FindBugs performed better than SRTA in both precision and recall for the L1 defect class, in addition to having slightly better precision in M1 and better recall in D2. The defect classes L4, L8, S1 and S2 were not supported by any of the tools. In case of SRTA it was due to the requirement of design involvement for L4 and L8, and the infeasibility of detecting S1 and S2 using static analysis. Classes L6, D4 and D5 were covered by SRTA alone. Thus, in comparison to the three other tools, SRTA is found to be covering more defect classes, with precision and recall higher than others in most cases, and close to the other tools in the few other cases.

To provide an additional view on the results, we have presented the data from Table 6.12 using an ROC-type plot as used by Zitser et al. [168]. The ROC-type plots, abbreviated from Receiver Operating Characteristics plots, are borrowed from wireless transmission analysis where they are used frequently to compare receiver peroformances. Following the procedure of Zitser et al. [168], we identified the Probability of Detection, P(d), and the Probability of False Alarms, P(f), as the following:

$$P(d) = \frac{DD}{N} \tag{6.9}$$

where,
DD = Defects detected correctly.
N = Total number of defects.

$$P(d) = \frac{ND}{N} \tag{6.10}$$

ND = Detection of non-defects as defects.
N = Total number of defects.

Table 6.12: Summary of Experimental Comparison of SRTA with UNO, SPLINT and FindBugs

		SRTA		UNO		FindBugs		SPLINT		
#	Class	Precision*	Recall*	Precision	Recall	Precision	Recall	Precision	Recall	Comment
1.	C1	0.94	0.93	-	-	1.00	0.90	-	-	
2.	C2	0.81	0.89	-	-	1.00	0.87	-	-	
3.	C3	0.92	0.84	0.75	0.66	0.67	0.79	-	-	
	C	0.89	0.89	0.75	0.66	0.89	0.85	-	-	
4.	L1	0.63	0.69	-	-	0.71	0.68	-	-	
5.	L2	0.77	0.87	-	-	-	-	-	-	
6.	L3	0.92	0.77	-	-	0.67	0.70	-	-	
7.	L4	-	-	-	-	-	-	-	-	Not supported
8.	L5	0.74	0.81	-	-	1.00	0.70	-	-	
9.	L6	0.88	0.81	-	-	-	-	-	-	
10.	L7	0.86	0.88			0.67	0.68	-	-	
11.	L8	-	-	-	-	-	-	-	-	Not supported
	L	0.80	0.81	-	-	0.79	0.69	-	-	
12.	M1	0.80	0.89	0.00	0.43	0.80	0.88	0.50	0.61	
13.	M2	0.81	0.88	-	-	-	-	/	0.57	Note 1
14.	M3	0.86	0.92	-	-	-	-	0.50	0.60	
15.	M4	0.77	0.92	0.25	0.00	-	-	0.38	0.51	
	M	0.81	0.90	0.13	0.22	0.8	0.88	0.46	0.57	
16.	D1	0.86	0.78	-	-	0.50	0.65	-	-	
17.	D2	0.36	0.77	-	-	0.33	0.79	-	-	
18.	D3	0.85	0.74	-	-	0.75	0.77	-	-	
19.	D4	0.65	0.87	-	-	-	-	-	-	
20.	D5	1.00	0.77	-	-	-	-	-	-	
	D	0.74	0.79	-	-	0.53	0.74	-	-	
21.	S1	-	-	-	-	-	-	-	-	Not supported
22.	S2	-	-	-	-	-	-	-	-	Not supported
	S	-	-	-	-	-	-	-	-	Not supported
	Average (over C, L, M, and D)	0.81	0.85	0.43	0.44	0.75	0.79	0.46	0.57	

*Lenient variant used
/ = Supported but data is not available
- = Not Supported

Note 1: Neither SPLINT nor SRTA detected any defect of class M2 in Sendmail.

As the same systems were used in both the precision and recall measurements, and the recall measurement systems only had one injected fault per system (repeated 1000 times), it can be assumed that the false positives were the same as they were for the precision assessment. In other words, although for the mutation experiment, we generated 1000 different variations of the same system with one defect injected to each, this approach was adopted only to facilitate accurate comparison. From a defect detector's point-of-view, injecting one fault in each of 1000 identical systems and injecting 1000 faults in one system have the same consequences.

Figure 6.3 shows the results presented in Table 6.12 in the ROC-type plot. SRTA's instance was marked as $SRTA_{LL}$ to indicate that the Lenient Variant of precision and Recall were used in this comparison. The horizontal axis shows the probability of false alarms, while the vertical shows the probability of detection. The diagonal line represents the probabilities where P(d) = P(f), signifying the performance of a purely random detection procedure. To be considered as a tool with reasonable performance, the performance of the tool has to be significantly above this random guessing line. To compute the significance, we have used

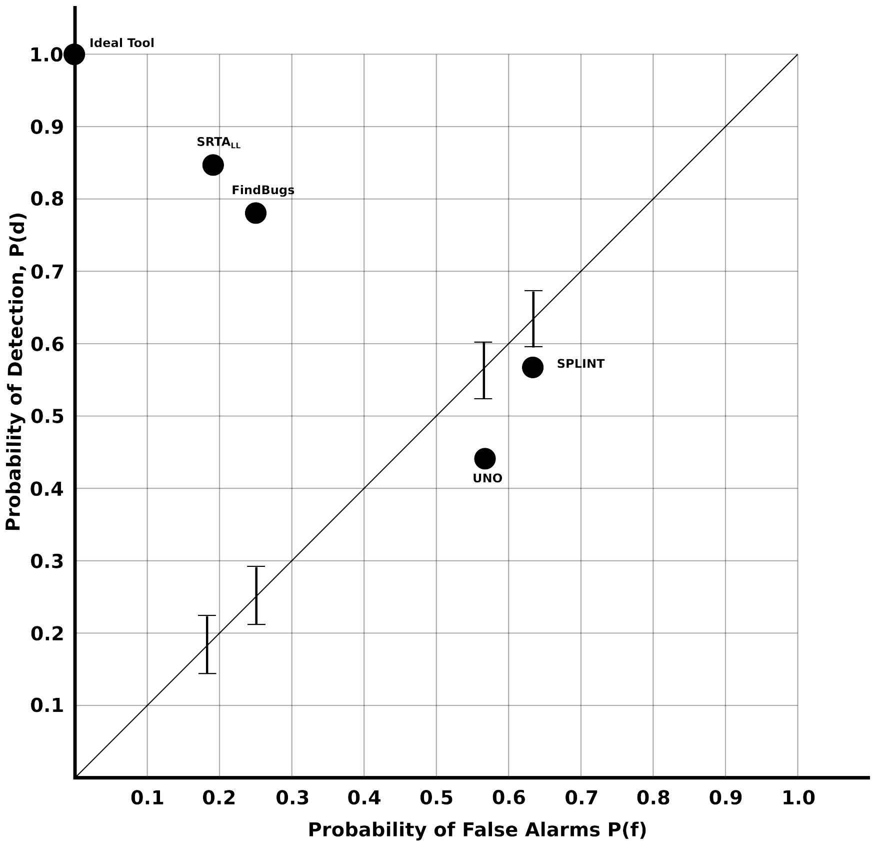


Figure 6.3: ROC-type Plot to Compare SRTA, UNO, FindBugs and SPLINT

the simple variance calculation, again following Zitser et al. [168]. The formula computes,

$$\sigma^2 = \frac{p(d)(1-p(d))}{N} \tag{6.11}$$

where,
σ^2 = The variance in estimation.
N = Total number of defects existing in the system.

The figure shows the significant distance using a $\pm\sigma$ error bar shown above and below the random guess line. The error bars are small in expanse due to the large sample set involved. The representation shows, that SPLINT, under our experiments, performed below the random guess line, and therefore is not considered a tool with significant performance. Same fact is observed for UNO. FindBugs and SRTA both perform significantly well above the line, with SRTA showing slightly better performance than FindBugs in both precision (as indicated by low false alarm rates) and recall (as indicated by high probability of detection).

Comparison Using Data from Literature

Table 6.13 shows the precision and recall data on the tools selected, along with their sources. In case a tool's precision or recall was reported by more than one sources, an average was taken into consideration.

Table 6.13: Comparison of SRTA's Precision with Other Tools Using Data from Literature

#	Tool/Author	Precision	Recall	Source	Comment
1.	$SRTA_{SS}$	0.65	0.76	This Experiment	Strict Precision and Recall
2.	$SRTA_{SL}$	0.65	0.85	This Experiment	Strict Precision
3.	$SRTA_{LS}$	0.81	0.76	This Experiment	Strict Recall
4.	$SRTA_{LL}$	0.81	0.85	This Experiment	
5.	ARCHER	0.56	0.01	[163, 168]	
6.	SPLINT	0.48	0.52	[50, 103, 168]	
7.	Marple	0.71	0.95	[97]	
8.	UNO	0.43	0.12	[103, 29, 168]	
9.	Parfait	1.00	0.75	[103]	
10.	Polyspace	0.67	0.94	[29, 168]	
11.	Coverity	0.80	0.48	[29]	

For SRTA, four different precision-recall sets were considered, using the four combinations of the lenient and strict variants of the precision and recall measurements. None of these tools, except Coverity, considers so many defect classes as SRTA. Most of them focuses on memory defects only. Comparing SRTA's performance in only the common defect classes with every tool would have guaranteed a better stand for SRTA, owing to the fact that the lower performance counts for SRTA comes from the logical defects, which none of these tools even consider. Still, instead of the common defects, the entire SRTA's performance was chosen to be compared with the entire performance of these tools to estimate the benefit of the technique as a whole.

The findings from Table 6.13 are presented in Figure 6.4 in another ROC-type plot. Although this representation differs with the previous ROC plot in one point - unlike the previous plot, a test of statistical significance could not be incorporated in this particular plot due to the secondary nature of the data, as the sample size could not be determined. Four versions of SRTA were presented in the table using the four variants of the precision and recall (over strict and lenient).

Intuitively, it can be said that the ideal tool should have the position (0.0, 1.0), showing zero false positives and detecting all defects that are present. Of the tools that have been compared, ARCHER and UNO clearly had performance below the random guess lines. SPLINT's characteristics was on the random guess line.

SRTA, Polyspace, Coverity, Parfait and Marple performed well above the random guess line. With Polyspace and Marple performing better than SRTA in terms of Recall. A closer look at Marple's evaluation [96] revealed its individual precision and recall measurement for different classes as, C1 = (0.80, 0.97), M1 = (0.8, 0.8), M3 = (0.5, 1.0) M4 = (0.87, 1.0), as it was reported by Le and Soffa [96]. This evaluation was obtained over Zitser's Benchmark [168] only. In comparison, assessing over a large-scale automated fault injection framework, on the BugBench Benchmark and on six industry applications, SRTA's corresponding precision and recall are found as C1 = (0.94, 0.93), M1 = (0.80, 0.89), M3 = (0.86, 0.92), M4 = (0.77, 0.92), as presented in earlier sections. The specific performance of SRTA was close with Marple in the specific common defect classes, although SRTA's evaluation was performed in a much larger scale. SRTA's average precision and recall were lowered by the performance in Logical and Data related defects, which Marple does not consider. This argument adds to SRTA's favour as an accurate tool.

For the tool Polyspace, little information is available on the detailed internal structure except that it is

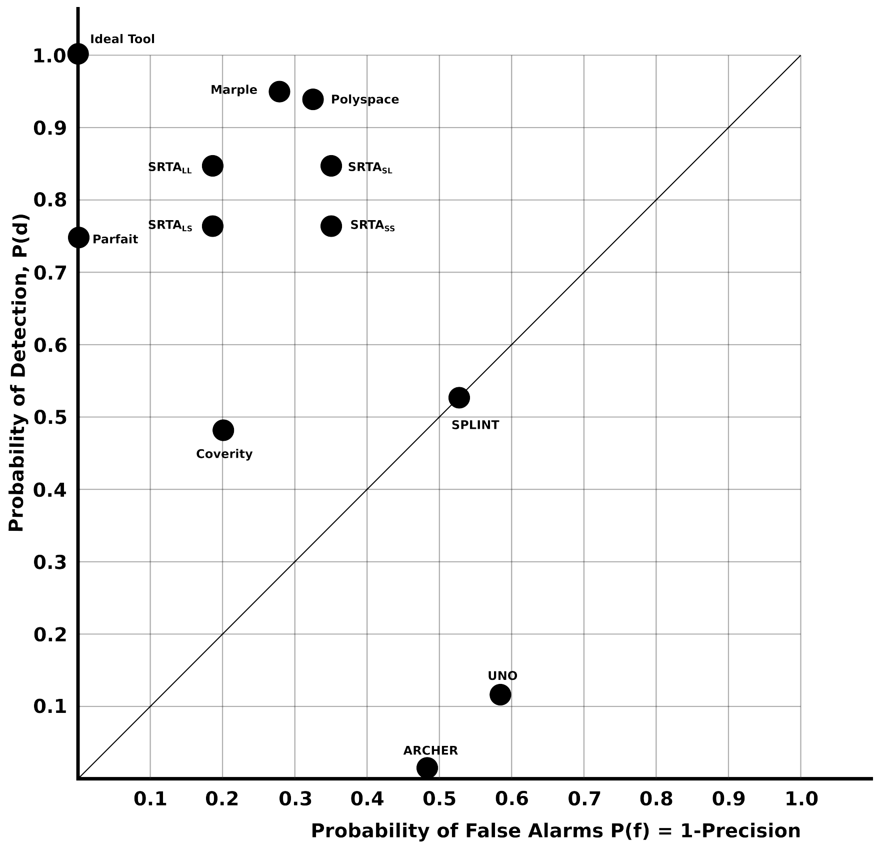


Figure 6.4: ROC-type Plot for Tool Performance Comparison

based on abstract interpretation and can detect C1, C3 and M4 defects [168, 29]. The average precision and recall, as reported by two studies [168, 29], were (0.75, 0.80), while against the same three defect classes, SRTA's average precision and recall are measured as (0.81, 0.85), almost at the same point as Polyspace, with a better recall. Like Marple, Polyspace does not consider the defect classes that lowered the average precision and recall of SRTA (i.e., Logic and Data related defects).

6.5.4 Practicality

Figure 6.5 shows the measures obtained from the controlled experiment. The graphs PMn, SMn, PMnC and Tn expresses the Primary Memory, Secondary Memory, Primary Memory Limit and Time, respectively, for System n. The scale in the vertical axis shows the memory units as multiples of 50MB and Time units as multiples of five minutes.

As apparent from the observations of the graphs, the system memory consumption, for both System 1 and System 2, are increasing with the increase of software size (although not in a linear trend). But as soon as the primary memory reaches the limiting capacity, the consumption on secondary memory increases more dramatically. This same pattern is displayed on both Systems. For TS3, System 1 reaches the limit but System 2 does not. As expected, System 1's secondary memory consumption graph matches the increasing

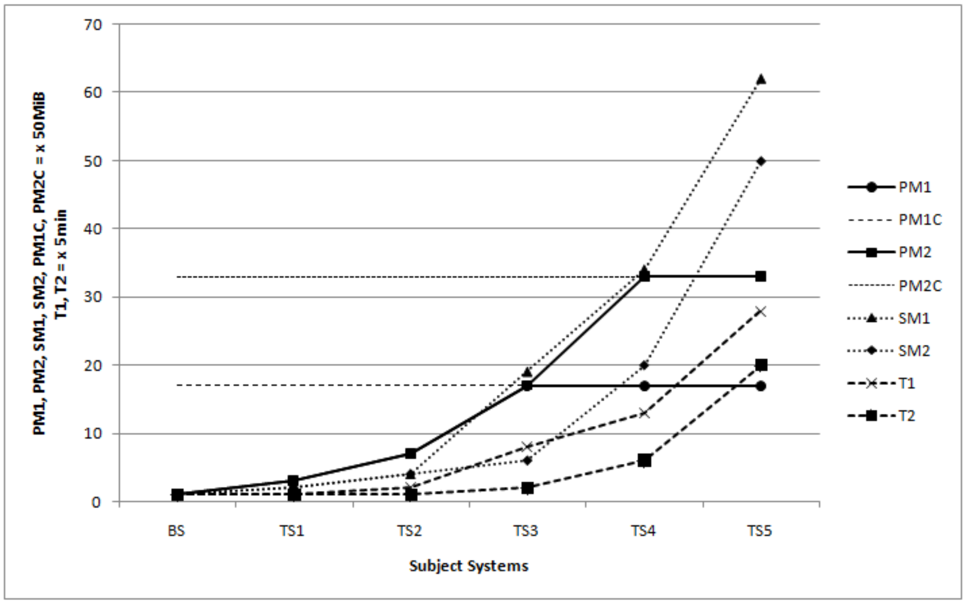


Figure 6.5: Different Performance Measures for Scalability and Practicality Assessment

requirement for secondary memory, while System 2's graph continues its previous gradual increment.

The graphs show that SRTA can balance itself on limiting resources, provided the secondary backup is sufficient. Also, as the processing power of System 1 is more limiting than System 2, the timing graph shows a more steep trend of increment. This observation can be substantiated into the conclusion that SRTA does not stop working for low-capacity systems, but requires more memory and time. On the other hand, using more processing power shall allow SRTA to run in less memory requirement.

6.6 Evaluation Outcome

Through direct experiment and the experiment involving the Mutation-Injection Framework, SRTA was found to exhibit high precision and recall, proving the point on its accuracy. In comparison with three other tools SRTA was found to be performing better than the others in most cases both in terms of precision and recall, and close to the others in the few exceptions. A controlled experiment verified SRTA's scalability and practicality, as well as all of the experiments establishing the argument in favour of its generality. If the outcome of all the experiments described in this chapter are combined together, the case for SRTA is established in that, it is highly accurate, scalable to systems of varying sizes, practical to perform within reasonably strict environmental constraints and general to detect multiple dissimilar classes of defects over widely varied test systems (by programming language, purpose, structure and complexity).

6.7 Answering the Research Questions

This section attempts to answer the research questions mentioned in Chapter 1. Answers to two research questions (RQ1 and RQ2), as provided in the previous chapter, are reinforced using experimental evidence in this section. In addition, this section answers two more research questions (RQ3 and RQ4).

6.7.1 RQ1: Detecting Multiple Dissimilar Classes of Defects

In answering the question “Can a specific abstraction provide sufficient means for detecting multiple dissimilar classes of defects?”, this chapter extends the statement provided by Chapter 5. Of the four experiments conducted in this research, three concerned on multiple defect classes.

As it is presented in Section 6.5, SRTA was able to detect 18 out of 22 classes of defects belonging to different similarity groups. The implementation worked entirely on the abstraction created by SRTA using the range tuples. The range tuples, and the four aspects of detection as were identifier by the previous chapter, was able to result into these multiple detections.

In summary, the experiments conducted proves three points - (a) SRTA is able to detect multiple dissimilar classes of defects, (b) the detection is carried out over a specific abstraction, instead of the source code, as outlined in Chapter 5, and (c) dissimilar defects can be represented by the same abstraction, proved by SRTA’s ability to detect them. When combined, the three points confirms the claim that it is possible to represent multiple dissimilar classes of defects over specific abstractions.

6.7.2 RQ2: Scope of Detection

The second research question, provided the first research question is answered affirmative, was stated as “To what length can such a technique go in terms of dissimilar classes of defects?”. This chaper provides the answer to this question by presenting the specific defect detection data. Of the 18 out of 22 defect classes SRTA detected, 13 defect classes were covered fully, as the defects belonging to these 13 classes were completely contained in the source code. The five other classes contained defects that, in some cases, are contained entirely in the source code but require architectural and design information in other cases and thus were partially covered. For the four classes of defects SRTA did not cover, two required runtime information and two required exclusive design information.

In summary, the experimental results described in this chapter confirms two points - (a) SRTA can detect the defects that are completely contained in source code in full, and (b) SRTA can sometimes detect defects that are not completely contained in source code, but whose footprint is prominent in source code. These two points collectively prove the hypothesis established in Chapter 5 that SRTA is able to detect defects that are contained in souce code.

6.7.3 RQ3: Accuracy and Performance

The third research question, provided the first research question is answered affirmative, was stated as, "What is the general effectiveness of such a technique in terms of accuracy and performance?". To answer this question, extensive experiments were conducted. The outcomes of the experiments are reported in this chapter.

Three of the four experiments reported the accuracy of detection in using different input set that contained six real world systems, 34000 mutated systems and one benchmark. Of all the detections, average precision was found to be 0.81 with the five-number statistical summary[1] for average precisions per defect class located in [0.36, 0.77, 0.83, 0.88, 1.00]. Average recall was found to be 0.85 with the five-number statistical summary for average recall per defect class located in [0.64, 0.77, 0.85, 0.89, 0.96]. The five-number summary expresses two qualities of the dataset. First, the more even are the distances among the the five components, the more evenly distributed are the data. Conversely, the less the distance among Q_1, Q_2 and Q_3 are from the even value, the more the data are clustered around a region. Second, if the data is clustered, the location of the Q_1, Q_2 and Q_3 determines the region of clustering. From the results presented in this section, it can be concluded that the data is clustered, and the clustering is towards the higer end of the spectrum.

To answer the research question, two points need to be considered from the data - (a) average precision and recall are high, as proven by the direct average and the five-point summary and (b) most of the data lie in the high end of the spectrum of all precisions and recalls measured - with 67% values over median for precision and 66% for recall. Both points considered together states that SRTA is a generally high performing tool for different defect classes, answering the first part of the research question.

To measure the performance, a controlled experiment was conducted. The performance was found to be, (a) almost linerarly increasing with the increase of code size, and (b) not limited by the primary memory of the computer SRTA runs on. The outcome of the controlled experiment confirmed SRTA to be a well performing tool under different environmental and input constraints, answering the second part of the research question.

6.7.4 RQ4: Effect of System Complexity

The fourth research question was stated as, provided a technique utilizing a specific abstraction to detect multiple dissimilar classes of defect exists, "What is the effect of system complexity on the applicability of such a Technique?". Relevant data to answer this question comes from all four of the experiments. SRTA was able to process systems with varying sizes - from small IDEs to the massive Linux Kernel. If the recall for specific systems are considered, SRTA was tested with 2000 mutated variants of two systems (Sendmail and BlueJ) that came from different domains (mailing demon and IDE), were implemented under different

[1]The five-number summary for a dataset D = $d_1, d_2, d_3, ...d_n$ is defined as the five-number set [$Limit_{min}$, Q_1, Q_2, Q_3, $Limit_{max}$]. $Limit_{min}$ and $Limit_{max}$ are the minimum and maximum values in the dataset. Q_1 is the value that segregates the dataset in two portions, with 25% of the data at or below it. Q_2 splits the dataset into half and Q_3 splits the dataset with 75% values below it. This particular number-collection is used to measure the bias in a dataset.

languages (C and Java), and used different technologies (Sendmail is a system demon, BlueJ is an application software). The different coefficients of variation of recall over these two systems using different defect classes were found to be in the range 0.05 to 0.07. Although the experiment to measure precision was not so extensive, it included six systems and the coefficient of variation of precision over the six systems for different defect classes were found to be in the range 0.08 to 0.21.

To answer the question, the specific point that needs to be considered is, the coefficients of variation are smaller than one and are closer to zero in the range [0, 1]. This observation signifies that the precision and recall did not vary widely over different system types for the same defect class. Therefore, it can be concluded that SRTA is able to retain its processing accuracy over varying system complexities.

6.8 Summary

This chapter provides the data and analyses from the experiments conducted to evaluate SRTA. The chapter started by establishing the mathematical framework used for evaluation and then described the four experiments conducted to evaluate SRTA. In the next section, the chapter described the evaluation outcome on the four perspectives - Accuracy, Scalability, Generality and Practicality. The analyses showed that SRTA is able to keep an average precision 0.81 and an average recall of 0.85, both high in comparison with the state-of-the-art, while generally applicable to multiple dissimilar defect classes, practical to work without special requirements or overhead and scalable to systems of arbitrary complexity. The data was provided in in this chapter in only the most relevant format required for establishing the points, the details are provided in Appendix C and B. The next chapter concludes the dissertation.

Chapter 7
Conclusion

This chapter contains the conclusive remarks to the dissertation, continuing the discussion from Chapters 3, 5, and 6. The chapter begins by describing the anticipated impacts of the research (Section 7.1), continues on describing the threats to validity of this research (Section 7.2) and finally describes the future work (Section 7.3)

7.1 Impact

This research demonstrated the possibility of using symbolic analysis, path summarization and state-space modelling through range analysis in the detection of multiple dissimilar classes of defects. The expectations from this research is to improve the trade of software maintenance, especially, the task of quality assurance. Subsequent sections specify the detailed outcome and expectations.

7.1.1 The Defect Model

Using symbolic analysis, path summaries and range analysis, a new symbolic model was built that captures the source code artifacts and their interactions. The model uses a three-tuple of symbolic range values derived from the state-space of the software. The 3-tuple signifies the three different phases of an entity's life cycle. The model provides a few key benefits,

(i) It is scalable to large systems, as it was demonstrated through the experiments in Chapter 6. The model successfully processed systems as large as the Linux Kernel.

(ii) The model is able to process incomplete code, as it is apparent from its construction. The model uses abstractions over source code that can ignore ambiguity at the code level. Two supportive cases were discovered while verifying false negatives in Chapter 6

(iii) It has Low False Positives, as it was demonstrated in the experiments described in Chapter 6

(iv) It has Low False Negatives, as it was demonstrated in the experiments described in Chapter 6

(v) It is Practical, as it was evident from the controlled experiment described in Chapter 6

(vi) It is General, as it was demonstrated by processing software systems implemented in three different languages (C, C++ and Java), and by comparison with other tools.

7.1.2 Multiple Defect Detection

Analysis techniques were developed to analyze the defect model to find anomalies as indications of defects. As the model is a simplified summarized representation, analysis techniques were free from the complexities associated with such detection, and thus became more manageable for the users.

The collective system of the model and the analysis techniques, specified under the name SRTA, was found to be accurate and scalable. A prototype tool under the same name was implemented and used in experiments.

7.1.3 FlexTax

As a prerequisite to this research, FlexTax was developed as a User-Driven Automated Framework for Defect Classification. FlexTax is aimed at balancing the human supervision and automated approach in taxonomy generation and is specifically designed to address the problems of inflexibility and non-extensibility in taxonomies. FlexTax utilizes the concepts from feature representation and comparison, and takes advantage of the accuracy of objective generation and mapping techniques, while providing for the flexibilities of human judgement through the user-driven nature.

Through a case study that involved more than 25000 real world defects, FlexTax was found to be capable of developing defect taxonomies that are complete, orthogonal, non-redundant, flexible and extensible. Additionally, FlexTax is a practical approach, as proven by the large number of defects it was able to map in a practically feasible time.

FlexTax can serve in the industry as it is and reduce maintenance overhead associated with defect mapping and taxonomy generation. FlexTax shifts the effort intensive tasks from the user to the automated system, allowing the mapping and / or remapping of a large number of defects in a fast and effective manner that can have a direct impact on the maintenance activities associated with re-creation of taxonomies.

7.1.4 The Defect Taxonomy

FlexTax was applied on the more than 25000 real world defect data as collected from CVE. The result was a defect taxonomy comprising of 27 defect classes organized in two hierarchical layers.

The taxonomy generated by FlexTax is generic in a sense that it was developed from the defect data coming from multiple projects that follow multiple languages, development paradigms, styles and structure. The taxonomy generated from this diverse set of defects, under the same formal framework as FlexTax, is thus applicable to general software defect scenarios.

The taxonomy can be readily adopted, adapted or extended as needed for the industry. This taxonomy is able to have a direct impact on reducing maintenance overhead as a ready solution to many software defect classification scenarios.

7.1.5 Defect Similarity

Using the underlying principles of FlexTax that rely on feature representation, different feature comparison metrics were compared to measure defect similarity. Although performed in the most basic form, these similarity metrics were able to provide different similarity measures towards the real-world defects. These similarity metrics can be applied as they are, and pave a direction for further development in the field.

7.2 Threats to Validity

- SRTA performs well above most tools, but there are a few specialized detection tools (as presented in Chapter 6) that outperform SRTA. This point needs to be considered from three aspects. First, although SRTA performs lower than some tools in specific cases, the general accuracy of SRTA is not low. Second, in the specific performance points where SRTA performed not as good as some other tools, SRTA did not perform significantly below the other tools. In fact, the difference was small for all cases. And Third, SRTA provides multiple defect detection capabilities which are not implemented by the other tools SRTA was compared against.

- A question may arise about the necessity of SRTA, where there are multiple specialized tools available for the task. If defect detection in software maintenance is carried out by a collection of specialized tools, two problems arise. First, the acquisition and maintenance of the multiple tools require maintenance overhead, and second, as different tools are different in their input specifications and requirements, they claim specialized human effort for their application. SRTA resolves both problems, as it covers the same defect types using one simple model.

- Another concern about the working principle of SRTA can be raised on the consideration of three different phases of an entity's life-cycle. Although it provides a better accuracy, as it was demonstrated in Chapter 6, there can be other intermediate or transient phases of an entity's life cycle that can have equally important effect on the detection. We recognize this question, and put the investigation on the topic as a future research direction.

- About the Taxonomy generated by FlexTax, which was the basis for the development for SRTA, a question may arise on the future processing capabilities of SRTA if the taxonomy is extended. In reality, SRTA's specification mechanism is soft-coded. As long as a new defect can be represented through static range analysis, SRTA can be configured to detect the defect.

7.3 Future Work

It is expected that the research presented in this dissertation shall be able to reduce maintenance overhead associated with the defect detection activity. Despite its rate of high accuracy, scalability, practicality and generality, there are multiple scopes for improvement for this research.

- The three-tuple used for SRTA has the capability to extend to multiple, but not all, defect types. The three-tuple signifies the three important phases of an entity's life cycle. The impact of other non-major phases of the entity's life cycle in its probability of being a defect is established, but extensive experiments on this topic can also be conducted. Research needs to be conducted into finding further extensions to the tuple, and to find whether additional phases, or especially, transition phases, can have an impact on the defects.

- SRTA, although general in specifications to process code from almost any statically-typed languages, was tested only on systems written in C++ and Java, further experiments can be performed to evaluate SRTA on systems developed using other industry adopted languages.

- This research specifies a defect taxonomy and evaluates the taxonomy against multiple procedures. The taxonomy, although is readily applicable to real-world systems, can be used to analyze defect impacts and distribution.

- This study provides introductory treatment to defect similarity assessment. The concept of similarity, as outlined in this dissertation, can be extended to large scale studies on defect similarity.

- The symbolic model developed and used in this research can be modified into detecting a challenging types of code clones, namely, the semantic clones [137]. A specific future plan for this research is to extend the model into developing a semantic clone detector, in collaboration with the state-of-the-art clone detectors like NiCad and SimCad [152, 151].

7.4 Additional Information

The SRTA prototype, the FlexTax prototype, data from the experiments, the specifications to use the prototypes, and other supplementary materials to this dissertation are provided through the website of the Software Research Lab of the University of Saskatchewan [11].

7.5 Summary

This chapter describes the expected impact, threats to validity and future works regarding the principles and techniques outlined in this dissertation in Chapters 3, 5 and 6. The chapter started with a statement of expected impact of FlexTax, SRTA, the Taxonomy and Defect Similarity in the current state-of-the-art. The chapter then described the specific problems encountered on the research, and the approach to their solution. Next, the chapter provided a summary of future works planned as continuation of this research. This Chapter concludes this dissertation.

References

[1] Albayrak, O., and Davenport, D. Impact of maintainability defects on code inspections. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2010), ESEM '10, ACM, pp. 50:1–50:4.

[2] Alkhatib, G. The maintenance problem of application software: an empirical analysis. *Journal of Software Maintenance 4*, 2 (June 1992), 83–104.

[3] Altova. Xml diffdog. http://www.altova.com/diffdog.html.

[4] Arumuga Nainar, P., Chen, T., Rosin, J., and Liblit, B. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 international symposium on Software testing and analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 5–15.

[5] Arumuga Nainar, P., and Liblit, B. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 255–264.

[6] Aslam, T. A taxonomy of security faults in the unix operating system. Master's thesis, Purdue University, august 1995.

[7] Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J., and Pugh, W. Using static analysis to find bugs. *Software, IEEE 25*, 5 (2008), 22–29.

[8] Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2007), PASTE '07, ACM, pp. 1–8.

[9] Bacchelli, A., and Bird, C. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 712–721.

[10] Basili, V. R., and Perricone, B. T. Software errors and complexity: an empirical investigation0. *Commun. ACM 27* (January 1984), 42–52.

[11] Billah, K., and Roy, C. Supporting materials for this dissertation. Available Online: `http://homepage.usask.ca/kab117/Data/ResearchData.html`.

[12] Bishop, M. A taxonomy of UNIX system and network vulnerabilities. Tech. Rep. CSE-9510, Department of Computer Science, University of California at Davis, May 1995.

[13] Bishop, M., and Bailey, D. A critical analysis of vulnerability taxonomies. Tech. Rep. CSE-96-11, Department of Computer Science, University of California at Davis, 1996.

[14] Blume, W., and Eigenmann, R. Symbolic range propagation. In *Proceedings of the 9th International Symposium on Parallel Processing* (Washington, DC, USA, 1995), IPPS '95, IEEE Computer Society, pp. 357–363.

[15] Blume, W., and Eigenmann, R. Demand-driven, symbolic range propagation. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, UK, 1996), LCPC '95, Springer-Verlag, pp. 141–160.

[16] Bond, M. D., Baker, G. Z., and Guyer, S. Z. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 13–24.

[17] Boshernitsan, M., Doong, R., and Savoia, A. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis* (New York, NY, USA, 2006), ISSTA '06, ACM, pp. 169–180.

[18] Brothers, L., Sembugamoorthy, V., and Muller, M. Icicle: groupware for code inspection. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work* (New York, NY, USA, 1990), CSCW '90, ACM, pp. 169–181.

[19] Budgen, D., and Brereton, P. Performing systematic literature reviews in software engineering. In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 1051–1052.

[20] Bush, W. R., Pincus, J. D., and Sielaff, D. J. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper. 30* (June 2000), 775–802.

[21] Cha, S.-H., Tappert, C. C., and Yoon, S. Enhancing binary feature vector similarity measures. *Journal of Pattern Recognition Research 1*, 1 (2006), 63–77.

[22] Chebaro, O., Kosmatov, N., Giorgetti, A., and Julliand, J. Combining static analysis and test generation for c program debugging. In *Proceedings of the 4th international conference on Tests and proofs* (Berlin, Heidelberg, 2010), TAP'10, Springer-Verlag, pp. 94–100.

[23] Chen, Z.-X., Zhan, J.-Y., and Hao, Z.-B. A new static pointer dereference detecting method based on finite-state machine. In *Apperceiving Computing and Intelligence Analysis (ICACIA), 2010 International Conference on* (dec. 2010), pp. 392 –397.

[24] Chen, Z.-X., Zhan, J.-Y., and Hao, Z.-B. A new static pointer dereference detecting method based on finite-state machine. In *Apperceiving Computing and Intelligence Analysis (ICACIA), 2010 International Conference on* (December 2010), pp. 392–397.

[25] Chess, B., and West, J. *Secure Programming with Static Analysis.* Pearson Education Ltd., 2007.

[26] Chilimbi, T. M., and Ganapathy, V. Heapmd: identifying heap-based bugs using anomaly detection. *SIGPLAN Not. 41* (October 2006), 219–228.

[27] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 34–44.

[28] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M.-Y. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering 18*, 11 (nov 1992), 943 –956.

[29] Chimdyalwar, B. Survey of array out of bound access checkers for c code. In *Proceedings of the 5th India Software Engineering Conference* (New York, NY, USA, 2012), ISEC '12, ACM, pp. 45–48.

[30] Choi, S.-S., Cha, S.-H., and Tappert, C. C. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics 8*, 1 (2010), 43–48.

[31] Chu, D.-H., and Jaffar, J. Symbolic simulation on complicated loops for wcet path analysis. In *Proceedings of the ninth ACM international conference on Embedded software* (New York, NY, USA, 2011), EMSOFT '11, ACM, pp. 319–328.

[32] Cifuentes, C., Hoermann, C., Keynes, N., Li, L., Long, S., Mealy, E., Mounteney, M., and Scholz, B. Begbunch: benchmarking for c bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)* (New York, NY, USA, 2009), DEFECTS '09, ACM, pp. 16–20.

[33] Cifuentes, C., Keynes, N., Li, L., and Scholz, B. Program analysis for bug detection using parfait: invited talk. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation* (New York, NY, USA, 2009), PEPM '09, ACM, pp. 7–8.

[34] Cordy, J. R., and Roy, C. K. The nicad clone detector. In *Proceedings of the Tool Demo Track of the 19th International Conference on Program Comprehension* (June 2011), ICPC 2011, IEEE Press, Kingston, Canada, pp. 219–220.

[35] Coverity Inc. Coverity SAVE. `http://www.coverity.com/products/coverity-save.html`.

[36] Csallner, C. *Combining over- and under-approximating program analyses for automatic software testing*. Ph.D., Georgia Tech, August 2008.

[37] Csallner, C., and Smaragdakis, Y. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper. 34* (September 2004), 1025–1050.

[38] Csallner, C., and Smaragdakis, Y. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 422–431.

[39] Csallner, C., Smaragdakis, Y., and Xie, T. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol. 17*, 2 (May 2008), 8:1–8:37.

[40] CTools Inc. The ctools libary. `http://sourceforge.net/projects/ctool/`.

[41] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods* (Berlin, Heidelberg, 2012), SEFM'12, Springer-Verlag, pp. 233–247.

[42] da Silva Villaca, R., de Paula, L., Pasquini, R., and Magalhaes, M. Hamming dht: Taming the similarity search. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE* (2013), pp. 7–12.

[43] Dallmeier, V., and Zimmermann, T. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 433–436.

[44] DeMillo, R. A., and Mathur, A. P. A grammar based fault classification scheme and its application to the classification of the errors of TeX. Tech. Rep. SERC-TR165-P, Software Engineering Research Center, Purdue University, September 1995.

[45] Dimitrov, M., and Zhou, H. Unified architectural support for soft-error protection or software bug detection. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (Washington, DC, USA, 2007), PACT '07, IEEE Computer Society, pp. 73–82.

[46] Duboc, L., Rosenblum, D., and Wicks, T. A framework for characterization and analysis of software system scalability. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 375–384.

[47] Ernst, M., Cockrell, J., Griswold, W., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on* (may 1999), pp. 213 –224.

[48] Evans, D. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (New York, NY, USA, 1996), PLDI '96, ACM, pp. 44–53.

[49] Evans, D., Guttag, J., Horning, J., and Tan, Y. M. Lclint: a tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1994), SIGSOFT '94, ACM, pp. 87–96.

[50] Evans, D., and Larochelle, D. Improving security using extensible lightweight static analysis. *Software, IEEE 19*, 1 (January 2002), 42 –51.

[51] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 234–245.

[52] Frama-C. Frama-c. `http://frama-c.com/what_is.html`.

[53] Freimut, B. Developing and using defect classification schemes. Tech. Rep. IESE-Report 072.01/E, Fraunhofer Institut für Experimentelles Software Engineering, 2001.

[54] Freimut, B., Denger, C., and Ketterer, M. An industrial case study of implementing and validating defect classification for process improvement and quality management. In *Software Metrics, 2005. 11th IEEE International Symposium* (sept. 2005), pp. 10–19.

[55] Freimut, B., Klein, B., Laitenberger, O., , and Ruhe, G. Experiencepackage from the essi process improvement experiment hyper. Tech. Rep. IESE-Report 015.00/E, Fraunhofer Institut für Experimentelles Software Engineering, 2000.

[56] Ganapathy, V., Jha, S., Chandler, D., Melski, D., and Vitek, D. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 345–354.

[57] Geay, E., Yahav, E., and Fink, S. Continuous code-quality assurance with safe. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 2006), PEPM '06, ACM, pp. 145–149.

[58] GNU. The fast lexical analyzer generator. `http://flex.sourceforge.net/`.

[59] Godefroid, P. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 47–54.

[60] Godefroid, P., and Luchaup, D. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 23–33.

[61] Goutte, C., and Gaussier, E. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *Proceedings of the 27th European conference on Advances in Information Retrieval Research* (Berlin, Heidelberg, 2005), ECIR'05, Springer-Verlag, pp. 345–359.

[62] Grady, R. B. *Practical Software Metrics For Project Management and Process Improvement.* Hewlett-Packard, 1992.

[63] Grammatech Inc. Codesonar. `http://www.grammatech.com/products/codesonar/overview.html`.

[64] Grammatech Inc. Codesonar price list. `http://www.grammatech.com/products/codesonar/pricelist.html`.

[65] Gray, J. Why do computers stop and what can be done about it? *Office 3* (June 1985).

[66] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The weka data mining software: an update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[67] Hamming, R. V. Error detecting and error correcting codes. *Bell Systems Technical Journal 29* (1950), 147–160.

[68] Holzmann, G. J. Static source code checking for user-defined properties. In *Proc IDPT 2002* (Pasadena, CA, USA, 2002).

[69] Hovemeyer, D., and Pugh, W. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2004), OOPSLA '04, ACM, pp. 132–136.

[70] Hovemeyer, D., and Pugh, W. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2007), PASTE '07, ACM, pp. 9–14.

[71] Hovemeyer, D., Spacco, J., and Pugh, W. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2005), PASTE '05, ACM, pp. 13–19.

[72] Howard, J. D. *An analysis of security incidents on the Internet 1989-1995.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1998. UMI Order No. GAX98-02539.

[73] IEEE. IEEE standard dictionary of measures to produce reliable software. *IEEE Standard 982.1-1988* (1989), 0–1.

[74] Jiang, L., and Su, Z. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 184–193.

[75] Jiang, L., and Su, Z. Profile-guided program simplification for effective testing and analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 48–58.

[76] Jiang, L., Su, Z., and Chiu, E. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 55–64.

[77] Jin, G., Thakur, A., Liblit, B., and Lu, S. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 241–255.

[78] Kelly, D., and Shepard, T. Qualitative observations from software code inspection experiments. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research* (2002), CASCON '02, IBM Press, pp. 5–.

[79] Kester, D., Mwebesa, M., and Bradbury, J. How good is static analysis at finding concurrency bugs? In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on* (September 2010), pp. 115–124.

[80] Kitchenham, B., Pearl Brereton, O., Budgen, D., Turner, M., Bailey, J., and Linkman, S. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol. 51* (January 2009), 7–15.

[81] Klocwork Inc. Klocwork insight. `http://www.klocwork.com/products/insight/?source=feature`.

[82] Knuth, D. E. The errors of tex. *Softw. Pract. Exper. 19*, 7 (July 1989), 607–685.

[83] Kola, G., Kosar, T., and Livny, M. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing* (Washington, DC, USA, 2004), GRID '04, IEEE Computer Society, pp. 251–258.

[84] Kong, D., Zheng, Q., Chen, C., Shuai, J., and Zhu, M. Isa: a source code static vulnerability detection system based on data fusion. In *Proceedings of the 2nd international conference on Scalable information systems* (ICST, Brussels, Belgium, Belgium, 2007), InfoScale '07, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 55:1–55:7.

[85] Krill, P. Klocwork tunes software code analysis suite for agile projects. `http://www.infoworld.com/d/developer-world/klocwork-tunes-software-code-analysis-suite-agile-projects-401`, 2009.

[86] Krill, P. Coverity offers integrity control to help manage code quality. `http://www.infoworld.com/d/application-development/coverity-offers-integrity-control-help-manage-code-quality-016?source=footer`, 2011.

[87] Krsul, I. V. *Software vulnerability analysis*. PhD thesis, Purdue University, West Lafayette, IN, USA, 1998. AAI9900214.

[88] Lal, S., and Sureka, A. A static technique for fault localization using character n-gram based information retrieval model. In *Proceedings of the 5th India Software Engineering Conference* (New York, NY, USA, 2012), ISEC '12, ACM, pp. 109–118.

[89] Landwehr, C. E., Bull, A. R., McDermott, J. P., and Choi, W. S. A taxonomy of computer program security flaws. *ACM Comput. Surv. 26* (September 1994), 211–254.

[90] Lanubile, F., Shull, F., and Basili, V. Experimenting with error abstraction in requirements documents. In *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International* (nov 1998), pp. 114 –121.

[91] Larson, E. Assessing work for static software bug detection. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007* (New York, NY, USA, 2007), WEASELTech '07, ACM, pp. 7–12.

[92] Larson, E. Suds: An infrastructure for creating bug detection tools. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 123–132.

[93] Le, W. *Toward a Practical, Path-Based Framework for Detecting and Diagnosing Software Faults*. PhD thesis, University of Virginia, 2010.

[94] Le, W., and Soffa, M. L. Refining buffer overflow detection via demand-driven path-sensitive analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2007), PASTE '07, ACM, pp. 63–68.

[95] Le, W., and Soffa, M. L. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 272–282.

[96] Le, W., and Soffa, M. L. Path-based fault correlations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 307–316.

[97] Le, W., and Soffa, M. L. Generating analyses for detecting faults in path segments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 320–330.

[98] Lehman, M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE special issue on Software Engineering 68*, 9 (sept. 1980), 1060 – 1076.

[99] Leszak, M., Perry, D. E., and Stoll, D. Classification and evaluation of defects in a project retrospective. *J. Syst. Softw. 61* (April 2002), 173–187.

[100] Leveson, N. G. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets 41* (2004), 564–575.

[101] Leveson, N. G., and Turner, C. S. An investigation of the therac-25 accidents. *Computer 26*, 7 (jul 1993), 18–41.

[102] Levine, J. *Flex and Bison.* O'Reilly Media, 2009.

[103] Li, L., Cifuentes, C., and Keynes, N. Practical and effective symbolic analysis for buffer overflow detection. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 317–326.

[104] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. Cp-miner: finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on 32*, 3 (2006), 176–192.

[105] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. *SIGPLAN Not. 40*, 6 (June 2005), 15–26.

[106] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 15–26.

[107] Lindqvist, U., and Jonsson, E. How to systematically classify computer security intrusions. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on* (may 1997), pp. 154 –163.

[108] Liu, C., and Han, J. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), SIGSOFT '06/FSE-14, ACM, pp. 46–56.

[109] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. Sober: statistical model-based bug localization. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ESEC/FSE-13, ACM, pp. 286–295.

[110] Lloyd, R. Metric mishap caused loss of nasa orbiter. `http://www.cnn.com/TECH/space/9909/30/mars.metric.02/index.html?_s=PM:TECH`, 1999.

[111] Lough, D. L. *A taxonomy of computer attacks with applications to wireless networks.* PhD thesis, Virginia Polytechnic Institute and State University, 2001. AAI3006082.

[112] Lu, S., Zhou, P., Liu, W., Zhou, Y., and Torrellas, J. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 38–52.

[113] Lucia, B., Wood, B. P., and Ceze, L. Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 378–388.

[114] Ma, L., and Tian, J. Analyzing errors and referral pairs to characterize common problems and improve web reliability. In *Proceedings of the 2003 international conference on Web engineering* (Berlin, Heidelberg, 2003), ICWE'03, Springer-Verlag, pp. 314–323.

[115] Ma, L., and Tian, J. Web error classification and analysis for reliability improvement. *J. Syst. Softw. 80* (June 2007), 795–804.

[116] Mariani, L. A fault taxonomy for component-based software. In *International Workshop on Test and Analysis of Component-Based Systems* (2003), vol. 83, pp. 55–65.

[117] McGraw, G. *Software Security: Building Security In.* Addision-Wesley, 2006.

[118] Mian, P., Conte, T., Natali, A., Biolchini, J., and Travassos, G. A Systematic Review Process for Software Engineering. In *ESELAW '05: 2nd Experimental Software Engineering Latin American Workshop* (2005).

[119] Mitre Corporation. Common vulnerabilities and exposure. `http://cve.mitre.org`.

[120] Nakamura, T., Hochstein, L., and Basili, V. R. Identifying domain-specific defect classes using inspections and change history. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (New York, NY, USA, 2006), ISESE '06, ACM, pp. 346–355.

[121] Ngo, M. N., and Tan, H. B. K. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ESEC-FSE '07, ACM, pp. 215–224.

[122] Nikora, A. P. *Software system defect content prediction from development process and product characteristics.* PhD thesis, University of Southern California, Los Angeles, CA, USA, 1998. AAI9902853.

[123] Ordonez, M. J., and Haddad, H. M. The state of metrics in software industry. In *Proceedings of the Fifth International Conference on Information Technology: New Generations* (Washington, DC, USA, 2008), ITNG '08, IEEE Computer Society, pp. 453–458.

[124] Ostrand, T. J., and Weyuker, E. J. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software 4* (1984), 289–300.

[125] Parnin, C., Görg, C., and Nnadi, O. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM symposium on Software visualization* (New York, NY, USA, 2008), SoftVis '08, ACM, pp. 77–86.

[126] Parnin, C., and Orso, A. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 199–209.

[127] PathCrawler. Pathcrawler. `http://pathcrawler-online.com/doDocumentation`.

[128] Ploski, J., Rohr, M., Schwenkenberg, P., and Hasselbring, W. Research issues in software fault categorization. *SIGSOFT Softw. Eng. Notes 32*, 6 (November 2007).

[129] Polyspace Inc. Polyspace embedded software verification. `http://www.mathworks.com/products/polyspace/`.

[130] Prause, C. R., and Apelt, S. An approach for continuous inspection of source code. In *Proceedings of the 6th international workshop on Software quality* (New York, NY, USA, 2008), WoSQ '08, ACM, pp. 17–22.

[131] Prause, C. R., and Eisenhauer, M. Social aspects of a continuous inspection platform for software source code. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering* (New York, NY, USA, 2008), CHASE '08, ACM, pp. 85–88.

[132] Rössler, J. Understanding failures through facts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software ngineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 404–407.

[133] Roy, C. K. Detection and analysis of near-miss software clones. In *Proceedings of the Doctoral Symposium Track of the 25th IEEE International Conference on Software Maintenance* (September 2009), ICSM 2009, pp. 447–450.

[134] Roy, C. K., and Cordy, J. R. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension* (June 2008), ICPC 2008, IEEE Press, Amsterdam, The Netherlands, pp. 172–181.

[135] Roy, C. K., and Cordy, J. R. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *Proceedings of the Poster Paper Track of the Canadian Conference on Computer Science and Software Engineering* (May 2008), C3S2E 2008, ACM Press, Montreal, Canada, pp. 137–140.

[136] Roy, C. K., and Cordy, J. R. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2009), ICSTW '09, IEEE Computer Society, pp. 157–166.

[137] Roy, C. K., Cordy, J. R., and Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. 470–495.

[138] Roychowdhury, S., and Khurshid, S. Software fault localization using feature selection. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering* (New York, NY, USA, 2011), MALETS '11, ACM, pp. 11–18.

[139] Saha, R. Detection and analysis of near miss clone genealogies. Master's thesis, University of Saskatchewan, august 2011.

[140] Seaman, C. B., Shull, F., Regardie, M., Elbert, D., Feldmann, R. L., Guo, Y., and Godfrey, S. Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (New York, NY, USA, 2008), ESEM '08, ACM, pp. 149–157.

[141] Shahriar, H., and Zulkernine, M. Classification of static analysis-based buffer overflow detectors. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on* (june 2010), pp. 94–101.

[142] Shukla, R., and Misra, A. K. Estimating software maintenance effort: a neural network approach. In *Proceedings of the 1st India software engineering conference* (New York, NY, USA, 2008), ISEC '08, ACM, pp. 107–112.

[143] Siddiqui, J. H., and Khurshid, S. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2012), OOPSLA '12, ACM, pp. 523–536.

[144] Sinha, N. Modular bug detection with inertial refinement. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Austin, TX, 2010), FMCAD '10, FMCAD Inc, pp. 199–206.

[145] Sokal, R. R., and Michener, C. D. A statistical method for evaluating systematic relationships. *University of Kansas Scientific Bulletin 28* (1958), 1409–1438.

[146] Stephenson, A. G., LaPiana, L. S., Mulville, D. R., Rutledge, P. J., Bauer, F. H., Folta, D., Dukeman, G. A., and Sackheim, R. Mars climate orbiter mishap investigation board phase i report. *Mars Climate Orbiter Mishap Investigation Board Phase I Report* (1999).

[147] Suhendra, V., Mitra, T., Roychoudhury, A., and Chen, T. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference* (New York, NY, USA, 2006), DAC '06, ACM, pp. 358–363.

[148] Svajlenko, J., Roy, C., and Cordy, J. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of Short/Tool Papers Track of the ICSE 7th International Workshop on Software Clones* (May 2013), IWSC 2013, pp. 8–9.

[149] Taghdiri, M., and Jackson, D. Inferring specifications to detect errors in code. *Automated Software Engg. 14* (March 2007), 87–121.

[150] Trainin, E., Nir-Buchbinder, Y., Tzoref-Brill, R., Zlotnick, A., Ur, S., and Farchi, E. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (New York, NY, USA, 2009), PADTAD '09, ACM, pp. 7:1–7:6.

[151] Uddin, S., Roy, C. K., and Schneider, K. Simcad : An extensible and faster clone detection tool for large scale software systems. In *Proceedings of the Tool Demonstration Track of the 21st IEEE International Conference on Program Comprehension* (May 2013), ICPC 2013, pp. 236–238.

[152] Uddin, S., Roy, C. K., Schneider, K. A., and Hindle, A. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of the 18th IEEE Working Conference on Reverse Engineering* (October 2011), WCRE 2011, IEEE Press, Lero, Limerick, Ireland, pp. 13–22.

[153] Vallespir, D., Grazioli, F., and Herbert, J. A framework to evaluate defect taxonomies. In *Proceedings of the CACIC 2009* (October 2009), pp. 643–652.

[154] Viega, J., Bloch, J. T., Kohno, T., and McGraw, G. Token-based scanning of source code for security problems. *ACM Trans. Inf. Syst. Secur. 5* (August 2002), 238–261.

[155] Vijayaraghavan, G. A taxonomy of e-commerce risks and failures. Master's thesis, Florida Institute of Technology, the USA, 2003.

[156] Vijayaraghavan, G., and Kaner, C. Bug taxonomies: Use them to generate better tests. In *Software Testing, Analysis and Review Conference (STAR) East* (2003).

[157] Vipindeep, V., and Jalote, P. Efficient static analysis with path pruning using coverage data. *SIGSOFT Softw. Eng. Notes 30*, 4 (May 2005), 1–6.

[158] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium* (2000), pp. 3–17.

[159] Wagner, S. Defect classification and defect types revisited. In *Proceedings of the 2008 workshop on Defects in large software systems* (New York, NY, USA, 2008), DEFECTS '08, ACM, pp. 39–40.

[160] Walden, J., Messer, A., and Kuhl, A. Idea: Measuring the effect of code complexity on static analysis results. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems* (Berlin, Heidelberg, 2009), ESSoS '09, Springer-Verlag, pp. 195–199.

[161] Ward, W. T. Calculating the real cost of software defects. *Hewlett-Packard Journal* (October 1991), 55–58.

[162] Weber, S., Karger, P. A., and Paradkar, A. A software flaw taxonomy: aiming tools at security. In *Proceedings of the 2005 workshop on Software engineering for secure systems - building trustworthy applications* (New York, NY, USA, 2005), SESS '05, ACM, pp. 1–7.

[163] Xie, Y., Chou, A., and Engler, D. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ESEC/FSE-11, ACM, pp. 327–336.

[164] Zhang, B., and Srihari, S. N. Properties of binary vector dissimilarity measures. In *Proc. JCIS Intl Conf. Computer Vision, Pattern Recognition, and Image Processing* (2003), vol. 1.

[165] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning* (New York, NY, USA, 2006), ICML '06, ACM, pp. 1105–1112.

[166] Zhou, B., Kulkarni, M., and Bagchi, S. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *Proceedings of the 20th international symposium on High performance distributed computing* (New York, NY, USA, 2011), HPDC '11, ACM, pp. 85–96.

[167] Zhuang, X., Zhang, T., and Pande, S. Using branch correlation to identify infeasible paths for anomaly detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 113–122.

[168] Zitser, M., Lippmann, R., and Leek, T. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (New York, NY, USA, 2004), SIGSOFT '04/FSE-12, ACM, pp. 97–106.

Appendix A

Taxonomy Framework Parameters

This appendix provides a list of the parameters used in generating the taxonomy in Chapter 3.

A.1 Perspectives

Unlike most defect repositories, CVE data is reported by personnel directly connected to the development and usage of the software. The reporters include systems analysis personnel, software engineers, security analysts, programmers and other personnel connected to the development of the system, or in its use in different places. These defect data are interpreted by people of the same trade. This unique nature of CVE makes it a repository where developers communicate defect data to the developers.

Despite being more technically oriented than general user groups, developers acting as users have the same limitation for closed source systems, and often with open source ones. Their communication tends to be using a black-box approach. The interpreters, however, have white-box access to the components.

A.2 Perspective 1

To model the reporting entities, the first perspective is chosen as the one describing the affiliation of the defect to the most visible major software component.

This perspective was chosen to consider CVE's defect reporters, who are usually technical professionals like System Analysts, Programmers or Security Analysts. Their description becomes detailed in technicality, but is limited only to the visible phenomena as the reporters do not usually have access to code.

A.3 Perspective 2

The second perspective was chosen as the one describing the exact technical issue that acted as the root cause for the defect.

This perspective was chosen to model the descriptions of CVE's developers who interpret the defect information as reported by the reporters. Being the professionals with access to the code, the developers are able to use more precise description of the defect with its detailed technical cause.

Table A.1: List of Attributes and Their Compliance to Defect Classes Developed in Chapter 3

			P1					P2																						
								Comp.			Logic								Memory				Data					Sync.		
#	Attribute	Weight	C	L	M	D	S	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	M1	M2	M3	M4	D1	D2	D3	D4	D5	S1	S2	Comment
1.1.	Wrong Output	10	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
1.2.	Wrong Format	20	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
1.3.	Memory Error	30	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
1.4.	Involves multiple entities	40	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
1.5.	Involves multiple processes	50	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.1	Truncated Values	5	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	
2.2.	Expanded Values	1	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	
2.3.	Rounded Values	5	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	
2.4.	Specific Offset	10	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	◐	◐	○	○	○	○	○	○	
2.5.	Unpredictable Values	20	○	○	○	○	○	○	○	●	○	○	○	◐	○	○	○	○	◐	○	○	◐	○	◐	○	○	○	○	○	
2.6.	Garbage Values	1	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	◐	○	○	○	○	◐	○	○	○	○	○	
2.7.	Missed a check	5	○	○	○	○	○	○	○	○	●	○	○	◐	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	
2.8.	Wrong condition in an if	5	○	○	○	○	○	○	○	○	●	○	○	◐	○	○	○	○	◐	○	○	○	○	○	◐	○	○	○	○	
2.9.	Wrong terminal in a loop	20	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.10.	No terminal condition	1	○	○	○	○	○	○	○	○	◐	◐	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.11.	Wrong Operator	30	○	○	○	○	○	○	○	○	○	○	●	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.12.	Wrong precedence	1	○	○	○	○	○	○	○	○	○	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.13.	Wrong operand	1	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.14.	Failed to save	14	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.15.	Failed to update	14	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.16.	Failed to set	14	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.17.	Missed / misinterpreted relation	1	○	○	○	○	○	○	○	○	○	○	○	◐	◐	○	○	◐	○	○	○	○	◐	○	○	○	◐	○	○	
2.18.	Extra complex logic	1	○	○	○	○	○	○	○	○	◐	○	○	◐	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	
2.19.	Failed to terminate a loop	15	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	
2.20.	Terminated a loop early	15	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	
2.21.	Made a tautology/contradiction	15	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	●	○	○	○	○	○	○	○	◐	○	○	○	○	
2.22.	Made more than required checks	26	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.23.	Made invalid extra checks	26	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
2.24.	Didn’t catch an exception	16	○	○	○	○	○	○	○	○	○	○	○	◐	○	●	○	◐	○	○	○	○	○	○	○	○	○	○	○	
2.25.	Caught wrong exception	16	○	○	○	○	○	○	○	○	○	○	○	◐	○	●	○	◐	○	○	○	○	○	○	○	○	○	○	○	
2.26.	Caught exception, didn’t handle	16	○	○	○	○	○	○	○	○	○	○	○	◐	○	●	○	◐	○	○	○	○	○	○	○	○	○	○	○	
2.27.	Caught exception, handled wrong	16	○	○	○	○	○	○	○	○	○	○	○	◐	○	●	○	◐	○	○	○	○	○	○	○	○	○	○	○	
2.28.	Made never-taken branch	15	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	
2.29.	Made always-taken branch	15	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	
2.30.	Wrong connections	4	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	●	○	○	○	◐	○	○	
2.31.	Wrong interactions	1	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	◐	○	◐	○	○	
2.32.	Changed algorithm in code	43	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	
2.33.	Changed implementation	43	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	
2.34.	Access non-allocated memory	4	○	○	○	○	○	○	◐	◐	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	
2.35.	Access memory out of valid range	22	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	
2.36.	Access memory with wrong attitude	4	○	○	○	○	○	◐	○	◐	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	
2.37.	Deallocated deallocated memory	8	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	
2.38.	Deallocated unallocated memory	8	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	
2.39.	Tried improper deallocation	8	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	
2.40.	Failed to deallocated	17	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	
2.41	Failed to allocate	4	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	
2.42.	Deallocated parts	17	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	
2.43.	Allocated less than required	22	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	
2.44.	Wrong interface values	4	○	○	○	○	○	◐	◐	◐	○	○	○	◐	○	○	○	◐	○	○	○	○	●	○	○	○	○	○	○	
2.45.	Wrong interface types	4	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	●	○	○	○	○	○	○	
2.46.	Default interface values	4	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	●	○	◐	○	○	○	○	
2.47.	Use interface without enough data	8	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	●	◐	○	○	○	○	
2.48.	Assign non-matching data	8	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	●	◐	○	○	○	○	
2.49.	Cast up or down	17	◐	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	●	○	○	○	○	
2.50.	Changed data to meet requirements	17	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	◐	●	○	○	○	○	
2.51.	Extra inputs supplied	22	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	●	○	○	○	
2.52.	Insufficient input supplied	22	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	●	○	○	○	
2.53.	Wrong data format	8	○	○	○	○	○	◐	○	○	◐	○	○	◐	○	○	○	◐	○	○	○	○	○	●	◐	○	○	○	○	
2.54.	Wrong access to members	27	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	○	●	○	○	
2.55.	Failed to provide access	27	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	○	●	○	○	
2.56.	Deadlock	5	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	○	○	●	○	
2.57.	Race	5	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	○	○	●	○	
2.58.	Out of Sync	10	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	○	○	◐	●	
2.59.	Lock and release problem	10	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	◐	○	○	○	○	○	○	○	○	○	◐	●	

●= Compliant as an essential attribute
◐= Compliant as an optional attribute
○= Non-Compliant

Appendix B

Additional Mutation Operators

Table B.1: Mutation Operators for the Three Complex Defect Classes (D3, D4, D5)

#	Defect	mOP	Language	Element	Mutation Construct	Example	Comment
1.	D3	m_{D3}	C, C++, Java	TP	$< if >< (>< expr ><) >$	$if(x > 0)$	
				ST	$< expr >$ contains idn	$x > 0$	
				ST	$< type >< id >< (> ... < type >< id_n > ... <) >$	$void\ function(int\ x)$	
				MP	$< if >< (>< 1 ><) >$	$if(1)$	
D3: The mutation changes a validation of a function parameter inside the function. The first subject-to condition ensures that a parameter is validated, and the second condition ensures that it is done inside a function.							
2.	D4	m_{D4}	C, C++	TP	$< struct >< type1 >< \{> ... < type >< id_n > ... <\} >$		
				ST	$< type >< id >< (> ... < type1 >< id_1 > ... <) >$	$void\ Shuffle(Cards\ cards[])$	
				ST	$< expr >$ contains idn		
				MP	$// < type >< id_n >$		
3.	D4	m_{D4}	Java	TP	$< class >< type1 >< \{> ... < public >< type >< id_n > ... <\} >$		
				ST	$< type >< id >< (> ... < type1 >< id_1 > ... <) >$	$void\ Shuffle(Cards\ cards[])$	
				ST	$< expr >$ contains id_n		
				MP	$// < public >< type >< id_n >$		
D4: The mutation creates an insufficient input problem. The first subject-to condition ensures that a composite structure is passed as a parameter, with the second condition ensuring that the specific value from the parameter is being used inside the function.							
4.	D5	m_{D5}	C, C++	ST	$< type >< id1 >< (> ... <) >$	$int * function()$	
				ST	$< class >< id >< \{> ... < public :>< type >< id_1 >< (> ... <) > ... <\} >$		
				ST	$< class >< id >< \{... < private :> ... < type >< id > ... ><\} >$	$private :\ int\ x;$	
				MP	$< return >< id >$	$return\ x;$	Inject.
5.	D5	m_{D5}	Java	ST	$< public >< type >< id1 >< (> ... <) >$	$int * function()$	
				ST	$< private > ... < type1 >< id > ... >$	$private :\ Item\ x;$	
				MP	$< type1 >< id_2 ><=>< id >< return >< id_2 >$	Item y = x; return y;	Inject.
D5: The mutation injects an improper access to a member entity to outside entitites. For Java, the matter is tricky due to the security already enforced by its constructs. The reference to a mutable item is created using the assignment which was then used to return the values, exposing the private member to the outside.							
TP = Token Pattern MP = Mutation Pattern ST = Subject to the existence of (condition) <> = Token							

Table B.2: Mutation Operators Developed for C++

#	Cls	mOP	Mutation	Example	Comment
1.	C1	m_{C1}	TP $< id >< = >< num >< ; >$	$x = 10;$	
			MP $< id >< = >< num1 >< ; >$	$x = 10 + 99999999999999;$	$num1$ = a large number
2.	C2	m_{C2}	TP $< id >< OP >< num >$	$x < 100;$	OP $= \{<, >, <=, >=, != , ==\}$
			MP $< id >< OP >< num1 >< + >< num2 >$	$x <= 100 + 10;$	$num1 = [1, 10]$
3.	C3	m_{C3}	TP $< type >< id >< = >< num >< ; >$	$int\ x = 0;$	$type = int \mid float \mid double \mid long$
			MP $< type >< id >< ; >$	$int\ x;$	
4.	L1	m_{L1}	TP $< if >< (>< id >< == >< expr ><) >$	$if(id == 10)$	
			MP $< if >< (>< id >< = >< expr ><) >$	$if(id = 10)$	
5.	L2	m_{L2}	TP $< for >< (> \ldots < id >< OP1 >< num > \ldots <) >$	$for(x = 0; x < 100; x++)$	{OP1,OP2} $= \{<, >\},\{>, <\},$
			MP $< for >< (> \ldots < id >< OP2 >< num > \ldots <) >$	$for(x = 0; x > 100; x++)$	$\{<=, >=\},\{>=, <=\},\{<, <=\}$
6.	L3	m_{L3}	TP $< for >< (> \ldots < id >< OP1 ><) >$	$for(x = 0; x < 100; x++)$	{OP1,OP2} $= \{++, --\},\{--, ++\}$
			MP $< for >< (> \ldots < id >< OP2 ><) >$	$for(x = 0; x < 100; x--)$	
7.	L5	m_{L5}	TP N/A		
			MP $< type >< id >;$	$intx;$	
8.	L6	m_{L6}	TP N/A	N/A	Inject.
			MP $< try >< \{ >< throw >< new >< type >< \} >< catch >< (>< type1 >< id ><) >$	$try\{throw\ new\ MyException\}$ $catch(Exception\ e)$	
9.	L7	m_{L7}	TP $< if >< (> \ldots <) >$	$if(x < 100)$	
			MP $< if >< (1) >$	$if(1)$	
10.	M1	m_{M1}	TP $< id >< = >< new >< type >< [>< num ><] >< ; >$	$p = newchar[100];$	
			MP $< id >< = >< 0 >$	$p = 0;$	
11.	M2	m_{M2}	TP $< delete >< id >< ; >$	$delete\ p;$	
			MP $< delete >< id >< ; >< delete >< id >< ; >$	$delete\ p; delete\ p;$	
12.	M3	m_{M3}	TP $< delete >< id >< ; >$	$delete\ p;$	
			MP $< / >< / >< delete >< id >< ; >$	$//delete\ p;$	
13.	M4	m_{M4}	TP $< id >< [>< num ><] >$	$p[30];$	
			MP $< id >< [>< num >< + >< num1 ><] >$	p[30+10];	
14.	D1	m_{D1}	TP $< id >< (>< id_1 >< , >< id_2 > \ldots < , >< id_n ><) >$	$function(x, y, z);$	
			MP $< id >< (>< id_1 >< , >< id_1 > \ldots < , >< id_1 ><) >$	$function(x, x, x);$	
15.	D2	m_{D2}	TP $< id >< (> \ldots < num > \ldots <) >$	$function(10, 20, 30);$	
			MP $< id >< (> \ldots < num1 > \ldots <) >$	$function(10, 0, 30);$	

Cls = Defect Class
TP = Token Pattern
<> = Token
mOP = Mutation Operator
MP = Mutation Pattern

Table B.3: Mutation Operators Developed for Java

#	Cls	mOP	Mutation	Example	Comment
1.	C1	m_{C1}	TP $< id >< = >< num >< ; >$	$x = 10;$	
			MP $< id >< = >< num1 >< ; >$	$x = 10 + 99999999999999;$	$num1$ = a large number
2.	C2	m_{C2}	TP $< id >< OP >< num >$	$x < 100;$	OP = $\{<, >, <=, >=, != , ==\}$
			MP $< id >< OP >< num1 >< + >< num2 >$	$x <= 100;$	$num1 = [1, 10]$
3.	C3	m_{C3}	TP $< type >< id >< = >< num >< ; >$	$int\ x = 0;$	$type = int \vert float \vert double \vert long$
			MP $< type >< id >< ; >$	$int\ x;$	
4.	L1	m_{L1}	TP $< if >< (>< id >< == >< expr ><) >$	$if(id == 10)$	
			MP $< if >< (>< id >< = >< expr ><) >$	$if(id = 10)$	
5.	L2	m_{L2}	TP $< for >< (> ... < id >< OP1 >< num > ... <) >$	$for(x = 0; x < 100; x++)$	{OP1,OP2} = $\{<, >\}$,$\{>, <\}$,
			MP $< for >< (> ... < id >< OP2 >< num > ... <) >$	$for(x = 0; x > 100; x++)$	$\{<=, >=\}$,$\{>=, <=\}$,$\{<, <=\}$
6.	L3	m_{L3}	TP $< for >< (> ... < id >< OP1 ><) >$	$for(x = 0; x < 100; x++)$	{OP1,OP2} = $\{++, --\}$,$\{--, ++\}$
			MP $< for >< (> ... < id >< OP2 ><) >$	$for(x = 0; x < 100; x--)$	
7.	L5	m_{L5}	TP N/A		
			MP $< type >< id >;$	$intx;$	
8.	L6	m_{L6}	TP N/A	N/A	Inject.
			MP $< try >< \{ >< throw >< new >< type >< \} >< catch >< (>< type1 >< id ><) >$	$try\{throw\ new\ MyException\}$ $catch(Exception\ e)$	
9.	L7	m_{L7}	TP $< if >< (> ... <) >$	$if(x < 100)$	
			MP $< if >< (1) >$	$if(1)$	
10.	M1	m_{M1}	TP $< id >< = >< new >< id >< [>< num ><] >< ; >$	$p = newchar[100];$	
			MP $< id >< = >< 0 >$	$p = 0;$	
13.	M4	m_{M4}	TP $< id >< [>< num ><] >$	$p[30];$	
			MP $< id >< [>< num >< + >< num1 ><] >$	p[30+10];	
14.	D1	m_{D1}	TP $< id >< (>< id_1 >< , >< id_2 > ... < , >< id_n ><) >$	$function(x, y, z);$	
			MP $< id >< (>< id_1 >< , >< id_1 > ... < , >< id_1 ><) >$	$function(x, x, x);$	
15.	D2	m_{D2}	TP $< id >< (> ... < num > ... <) >$	$function(10, 20, 30);$	
			MP $< id >< (> ... < num1 > ... <) >$	$function(10, 0, 30);$	

Cls = Defect Class
TP = Token Pattern
<> = Token
mOP = Mutation Operator
MP = Mutation Pattern

Appendix C
Detailed Evaluation Data

Table C.1: Precision of SRTA in Processing the Test Systems

Detection Data																				
		Firefox			Thunderbird			Notepad++			Sendmail			Linux			BlueJ			
#	Class	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	Comment
1.	C1	8	2	2	11	2	2	2	0	0	3	1	0	21	5	2	2	0	0	
2.	C2	7	1	2	13	1	3	0	0	0	1	0	1	12	1	1	1	0	0	
3.	C3	2	0	0	17	1	4	1	1	0	5	0	1	14	2	3	2	1	0	
4.	L1	1	0	2	2	1	2	0	0	0	4	2	2	9	2	3	5	1	3	
5.	L2	1	1	2	0	0	0	0	0	0	0	0	0	7	1	2	1	0	0	
6.	L3	0	0	0	0	0	0	0	0	0	0	0	0	8	2	2	3	0	0	
7.	L5	0	0	0	3	0	2	2	0	1	2	0	1	28	0	9	1	1	0	
8.	L6	0	0	0	0	0	0	0	0	0	0	0	0	3	0	1	1	0	0	
9.	L7	2	0	1	1	0	0	1	0	0	0	0	0	0	0	0	2	1	1	
10.	M1	12	2	3	1	1	1	1	1	0	2	1	2	68	8	17	6	0	1	
11.	M2	2	0	1	1	0	0	0	0	0	0	0	0	3	0	1	-	-	-	
12.	M3	4	1	1	6	0	3	1	0	0	1	1	0	87	5	26	-	-	-	
13.	M4	6	0	2	3	0	1	0	0	0	3	1	1	102	12	46	4	1	1	
14.	D1	0	0	0	0	0	0	0	0	0	1	1	0	9	3	9	2	1	0	
15.	D2	0	0	0	0	0	0	0	0	2	0	0	1	7	5	1	1	0	1	
16.	D3	19	4	4	0	0	0	1	0	0	2	0	2	18	2	2	3	1	0	
17.	D4	1	0	1	0	0	0	0	0	0	0	0	0	4	0	1	0	0	0	
18.	D5	0	0	0	0	0	0	1	0	0	0	0	0	6	0	0	0	0	0	

Precision																				
		Firefox			Thunderbird			Notepad			Sendmail			Linux			BlueJ			
#	Class	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	P_S	P_L	F_P	Comment
1.	C1	0.67	0.83	0.17	0.73	0.87	0.13	1.00	1.00	0.00	0.75	1.00	0.00	0.75	0.93	0.07	1.00	1.00	0.00	
2.	C2	0.70	0.80	0.20	0.76	0.82	0.18	-	-	-	0.50	0.50	0.50	0.86	0.93	0.07	1.00	1.00	0.00	
3.	C3	1.00	1.00	0.00	0.77	0.82	0.18	0.50	1.00	0.00	0.83	0.83	0.17	0.74	0.84	0.16	0.67	1.00	0.00	
4.	L1	0.33	0.33	0.67	0.40	0.60	0.40	-	-	-	0.50	0.75	0.25	0.64	0.79	0.21	0.56	0.67	0.33	
5.	L2	0.25	0.50	0.50	-	-	-	-	-	-	-	-	-	0.70	0.80	0.20	1.00	1.00	0.00	
6.	L3	-	-	-	-	-	-	-	-	-	-	-	-	0.67	0.83	0.17	1.00	1.00	0.00	
7.	L5	-	-	-	0.60	0.60	0.40	0.67	0.67	0.33	0.67	0.67	0.33	0.76	0.76	0.24	0.50	1.00	0.00	
8.	L6	-	-	-	-	-	-	-	-	-	-	-	-	0.75	0.75	0.25	1.00	1.00	0.00	
9.	L7	0.67	0.67	0.33	1.00	1.00	0.00	1.00	1.00	0.00	-	-	-	-	-	-	0.50	0.75	0.25	
10.	M1	0.71	0.82	0.18	0.33	0.67	0.33	0.50	1.00	0.00	0.40	0.60	0.40	0.73	0.82	0.18	0.86	0.86	0.14	
11.	M2	0.67	0.67	0.33	1.00	1.00	0.00	-	-	-	-	-	-	0.75	0.75	0.25	-	-	-	
12.	M3	0.67	0.83	0.17	0.67	0.67	0.33	1.00	1.00	0.00	0.50	1.00	0.00	0.74	0.78	0.22	-	-	-	
13.	M4	0.75	0.75	0.25	0.75	0.75	0.25	-	-	-	0.60	0.80	0.20	0.64	0.71	0.29	0.67	0.83	0.17	
14.	D1	-	-	-	-	-	-	-	-	-	0.50	1.00	0.00	0.43	0.57	0.43	0.67	1.00	0.00	
15.	D2	-	-	-	-	-	-	0.00	0.00	1.00	0.00	0.00	1.00	0.54	0.92	0.08	0.50	0.50	0.50	
16.	D3	0.70	0.85	0.15	-	-	-	1.00	1.00	0.00	0.50	0.50	0.50	0.82	0.91	0.09	0.75	1.00	0.00	
17.	D4	0.50	0.50	0.50	-	-	-	-	-	-	-	-	-	0.80	0.80	0.20	-	-	-	
18.	D5	-	-	-	-	-	-	1.00	1.00	0.00	-	-	-	1.00	1.00	0.00	-	-	-	

CC = Detected under the Correct Class	P_S = Precision (Strict Estimation)
IC = Detected under Incorrect Class	P_L = Precision (Lenient Estimation)
ND = Detected, but not a defect	F_P = False Positive Rate

Table C.2: SRTA's Recall Assessment from the Mutation-Injection Experiment

				Detection Count						Recall						
				Sendmail			BlueJ			Sendmail			BlueJ			
#	Defect	mOP	Mutants	CC	IC	ND	CC	IC	ND	R_S	R_L	F_N	R_S	R_L	F_N	Comment
1.	C1	m_{C1}	1000	896	0	104	892	12	96	0.89	0.89	0.11	0.89	0.90	0.10	
2.	C2	m_{C2}	1000	856	52	92	857	17	126	0.86	0.91	0.09	0.86	0.87	0.13	
3.	C3	m_{C3}	1000	878	0	122	780	8	212	0.88	0.88	0.12	0.78	0.79	0.21	
4.	L1	m_{L1}	1000	619	112	269	634	12	354	0.62	0.73	0.27	0.63	0.64	0.36	
5.	L2	m_{L2}	1000	773	103	124	812	36	152	0.77	0.88	0.12	0.81	0.85	0.15	
6.	L3	m_{L3}	1000	711	81	208	732	21	257	0.71	0.79	0.21	0.73	0.75	0.25	
7.	L5	m_{L5}	1000	726	127	147	691	19	290	0.73	0.85	0.15	0.69	0.71	0.29	
8.	L6	m_{L6}	1000	667	187	146	751	18	231	0.67	0.85	0.15	0.75	0.77	0.23	
9.	L7	m_{L7}	1000	794	166	40	791	6	203	0.79	0.96	0.04	0.79	0.80	0.20	
10.	M1	m_{M1}	1000	815	68	117	896	2	102	0.82	0.88	0.12	0.90	0.90	0.10	
11.	M2	m_{M2}	1000	791	89	120	-	-	-	0.79	0.88	0.12	-	-	-	
12.	M3	m_{M3}	1000	839	76	85	-	-	-	0.84	0.92	0.08	-	-	-	
13.	M4	m_{M4}	1000	817	81	102	890	42	68	0.82	0.90	0.10	0.89	0.93	0.07	
14.	D1	m_{D1}	1000	709	98	193	729	10	261	0.71	0.81	0.19	0.73	0.74	0.26	
15.	D2	m_{D2}	1000	751	102	147	672	9	319	0.75	0.85	0.15	0.67	0.68	0.32	
16.	D3	m_{D3}	1000	618	162	220	599	100	301	0.62	0.78	0.22	0.60	0.70	0.30	
17.	D4	m_{D4}	1000	702	119	179	753	152	95	0.70	0.82	0.18	0.75	0.91	0.09	
18.	D5	m_{D5}	1000	686	139	175	594	103	303	0.69	0.83	0.17	0.59	0.70	0.30	
	Average		1000	758	98	144	782	32	187	0.76	0.86	0.14	0.78	0.81	0.19	

mOP = Mutation Operator	R_S = Recall (Strict Estimation)
CC = Detected under Correct Class	R_L = Recall (Lenient Estimation)
IC = Detected under Incorrect Class	F_N = False Negative
ND = Not Detected	

Table C.3: SRTA's Recall Assessment by Application on BugBench

Detection Data																						
	bc			cvs			gzip			man			ncompress			polymorph			squid			
# Class	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	CC	IC	ND	Comment
1. L3	-	-	-	-	-	-	-	-	-	1	0	0	-	-	-	-	-	-	-	-	-	
2. M2	-	-	-	1	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
3. M4	3	1	0	-	-	-	1	0	0	0	0	1	1	0	0	1	1	0	1	0	0	

Recall																						
	bc			cvs			gzip			man			ncompress			polymorph			squid			
# Class	R_C	R_L	F_N	R_C	R_L	F_N	R_C	R_L	F_N	R_C	R_L	F_N	R_C	R_L	F_N	R_C	R_L	F_N	R_C	R_L	F_N	Comment
1. L3	-	-	-	-	-	-	-	-	-	1.00	1.00	0.00	-	-	-	-	-	-	-	-	-	
2. M2	-	-	-	1.00	1.00	0.00	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
3. M4	0.75	1.00	0.00	-	-	-	1.00	1.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00	0.50	1.00	0.00	1.00	1.00	0.00	
Avg	0.75	1.00	0.00	1.00	1.00	0.00	0.50	1.00	0.00	0.50	0.50	0.50	1.00	1.00	0.00	0.50	1.00	0.00	1.00	1.00	0.00	

CC = Detected under correct class	R_C = Recall (Strict Estimation)
IC = Detected under Incorrect Class	R_L = Recall (Lenient Estimation)
ND = Not Detected	F_N = False Negative

Table C.4: Comparison of SRTA's Precision with UNO, FindBugs and SPLINT Using Experimental Data

					Tool						SRTA						
# Tool	System	Language	LOC	Class	CC	IC	ND	P_S	P_L	F_P	CC	IC	ND	P_S	P_L	F_P	Comment
1. UNO	Sendmail-8.12.11	C	102k	C3	3	0	1	0.75	0.75	0.25	5	0	1	0.83	0.83	0.17	SRTA: Table C.1
				M1	2	0	2	0.00	0.00	1.00	2	1	2	0.40	0.60	0.40	SRTA: Table C.1
				M4	4	0	3	0.25	0.25	0.75	3	1	1	0.60	0.80	0.20	SRTA: Table C.1
2. Findbugs	BlueJ-3.0.9	Java	86k	C1	2	0	0	1.00	1.00	0.00	2	0	0	1.00	1.00	0.00	
				C2	1	0	0	1.00	1.00	0.00	1	0	0	1.00	1.00	0.00	
				C3	2	0	1	0.67	0.67	0.33	2	1	0	0.67	1.00	0.00	
				L1	5	0	2	0.71	0.71	0.29	5	1	3	0.56	0.67	0.33	
				L3	2	0	1	0.67	0.67	0.33	3	0	0	1.00	1.00	0.00	
				L5	1	0	0	1.00	1.00	0.00	1	1	0	0.50	1.00	0.00	
				L7	2	0	1	0.67	0.67	0.33	2	1	1	0.50	0.75	0.25	
				M1	4	0	1	0.80	0.80	0.20	6	0	1	0.86	0.86	0.14	
				D1	2	0	2	0.50	0.50	0.50	2	1	0	0.67	1.00	0.00	
				D2	1	0	2	0.33	0.33	0.67	1	0	1	0.50	0.50	0.50	
				D3	3	0	1	0.75	0.75	0.25	3	1	0	0.75	1.00	0.00	
3. SPLINT	Sendmail-8.12.11	C	102k	M1	1	0	1	0.50	0.50	0.50	2	1	2	0.40	0.60	0.40	SRTA: Table C.1
				M2	0	0	0	-	-	-	0	0	0	-	-	-	None was present
				M3	3	0	3	0.50	0.50	0.50	1	1	0	0.50	1.00	0.00	SRTA: Table C.1
				M4	5	0	8	0.38	0.38	0.62	3	1	1	0.60	0.80	0.20	SRTA: Table C.1

CC = Detected under Correct Class P_S = Precision (Strict Estimation)
IC = Detected under Incorrect Class P_L = Precision (Lenient Estimation)
ND = Not a Defect F_P = False Positive

Table C.5: Comparison of SRTA's Recall with UNO, FindBugs and SPLINT Using Experimental Data

							Tool						SRTA						
# Tool	System	Lang.	LOC	Class	mOP	Count	CC	IC	ND	R_S	R_L	F_N	CC	IC	ND	R_S	R_L	F_N	Comment
1. UNO	Sendmail-8.12.11	C	102k	C3	m_{C3}	1000	661	0	339	0.66	0.66	0.34	896	0	104	0.88	0.88	0.12	SRTA: Table C.2
				M1	m_{M1}	1000	434	0	366	0.43	0.43	0.37	815	68	117	0.82	0.88	0.12	SRTA: Table C.2
				M4	m_{M4}	1000	5	0	1000	0.00	0.00	1.00	817	81	102	0.82	0.90	0.10	SRTA: Table C.2
2. FindBugs	BlueJ-3.0.9	Java	86k	C1	m_{C1}	1000	896	0	104	0.90	0.90	0.10	892	12	96	0.89	0.90	0.10	
				C2	m_{C2}	1000	871	0	129	0.87	0.87	0.13	857	17	126	0.86	0.87	0.13	
				C3	m_{C3}	1000	792	0	208	0.79	0.79	0.21	780	8	212	0.78	0.79	0.21	
				L1	m_{L1}	1000	681	0	319	0.68	0.68	0.32	634	12	354	0.63	0.64	0.36	
				L3	m_{L3}	1000	702	0	298	0.70	0.70	0.30	732	21	257	0.73	0.75	0.25	
				L5	m_{L5}	1000	698	0	302	0.70	0.70	0.30	751	18	769	0.69	0.71	0.29	
				L7	m_{L7}	1000	678	0	322	0.68	0.68	0.32	791	6	203	0.79	0.80	0.20	
				M1	m_{M1}	1000	887	0	113	0.88	0.88	0.12	896	2	102	0.90	0.90	0.10	
				D1	m_{D1}	1000	652	0	348	0.65	0.65	0.35	729	10	261	0.73	0.74	0.26	
				D2	m_{D2}	1000	791	0	209	0.79	0.79	0.21	672	9	319	0.67	0.68	0.32	
				D3	m_{D3}	1000	772	0	228	0.77	0.77	0.23	599	100	301	0.60	0.70	0.30	
3. SPLINT	Sendmail-8.12.11	C	102k	M1	m_{M1}	1000	609	0	391	0.61	0.61	0.39	815	68	117	0.82	0.88	0.12	SRTA: Table C.2
				M2	m_{M2}	1000	568	0	432	0.57	0.57	0.43	791	89	120	0.79	0.88	0.12	SRTA: Table C.2
				M3	m_{M3}	1000	602	0	398	0.60	0.60	0.40	839	76	84	0.84	0.92	0.08	SRTA: Table C.2
				M4	m_{M4}	1000	507	0	493	0.51	0.51	0.49	817	81	102	0.82	0.90	0.10	SRTA: Table C.2

CC = Detected under Correct Class R_S = Recall (Strict Estimation)
IC = Detected under Incorrect Class R_L = Recall (Lenient Estimation)
ND = Not a Defect F_N = False Negative