ABSTRACT

SHERRIFF, MARK STEPHEN. Analyzing Software Artifacts through Singular Value Decomposition to Guide Development Decisions. (Under the direction of Laurie A. Williams.)

During development, programming teams will produce numerous types of software development artifacts. A software development artifact is an intermediate or final product that is the result or by-product of software development. Hidden relationships and structures within a software system can be illuminated through singular value decomposition using software development artifacts, and these relationships can be leveraged to help guide software development questions regarding the interactions among software files.

The goal of this research is to build and investigate a framework called Software Development Artifact Analysis (SDAA) that uses software development artifacts to illuminate underlying relationships within a system. SDAA provides guidelines for selecting and gathering software development artifacts, discovering relationships, and then leveraging the insights gained through the analysis of those relationships. We use singular value decomposition (SVD) to generate the relationships from a matrix of software development artifact metrics.

In this research, we use SDAA to create three SVD-based software development analysis techniques: an impact analysis technique, a regression test prioritization technique, and a static analysis alert filtering technique. These techniques were applied and examined on an industrial project and five open source projects and compared with comparable current techniques. In general, our techniques were shown to be more resource efficient than comparable techniques in system resources and time, while still prioritizing developer effort effectively.

Analyzing Software Artifacts Through Singular Value Decomposition to Guide Development Decisions

by
Mark Stephen Sherriff

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2007

APPROVED BY:

Dr. Thomas L. Honeycutt	Dr. Jason A. Osborne
Dr. Mladen A. Vouk	Dr. Laurie A. Williams Chair of Advisory Committee

DEDICATION

To my wife, Amanda

BIOGRAPHY

Mark S. Sherriff was born November 9, 1979 in Salisbury, North Carolina. Mark grew up in Salisbury and Rock Hill, SC, and received his first computer (a 486 SX 25 Mhz) in the eighth grade. Since then, he's been fascinated by technology, computing, and how to make the experience better for everyone, from developers to end-users. Mark received his B.S. from Wake Forest University in Computer Science in 2002. In 2004, Mark earned his M.S. from North Carolina State University on the way to his Ph.D. in 2007.

As a technology enthusiast, Mark enjoys everything from building computers to web design to playing video games. He also enjoys cooking, traveling, and has been involved in handbell choirs for over seventeen years.

ACKNOWLEDGEMENTS

As any doctoral student will attest, the pursuit of a PhD cannot be done alone and most certainly can only come about with the help and support of many people. Over the past five years, Dr. Laurie Williams has been a continual source of wisdom, support, and encouragement for me. In a field that is constantly searching for women in every possible role from student to professor, I have had the fortune of working with one of the best. Words cannot adequately express the gratitude that I have for her seemingly unending source of patience, words of encouragement, and insights into software engineering. When I came to NC State, it is fair to say that I was a good student, but really had no inkling as to what my future chosen profession actually entailed. Laurie has been an exceptional mentor, a good friend, and the main academic reason that I am even writing this paragraph. I sincerely hope that she has gained as much from our time together as I have, and hopefully it is something other than my tendency to exaggerate greatly.

During my time at NC State, I also had the opportunity of working with a man who once told me that he appeared in numerous PhD acknowledgement sections as a person who gave a key insight. It didn't take me long to discover just why that was the case. Dr. J. Michael Lake gave me not only a key insight into this research; he also helped provide the means and amazing amounts of guidance. It is almost a certainty that without his assistance, this dissertation would be about something else completely and would have a date perhaps a year or two later. Mike's ability to dissect a problem and his tenacity at solving it served as an example to me during my time at IBM. To him, Jim Fletcher (my manager at IBM), and the rest of my team at IBM, I give my sincere thanks. IBM's support of research at NC State

University has been extremely important in the lives of numerous doctoral students. The atmosphere of support at IBM for students and research is refreshing and much appreciated.

In addition to the research support from IBM, several other individuals and organizations were generous in their research and financial support. The Center for Advanced Computing and Communication has been a continual source of financial support for my research and for others in my research group. Specifically, the support and encouragement of Brad Martin through the National Science Foundation has been of immense benefit to me over my five years at NCSU.

I would also like to thank my doctoral committee. My first real impression of Dr. Mladen Vouk was during my Master's examination, where he, while eating pastries I had made the night before, asked question after question, causing me to doubt nearly everything I showed. Afterwards, thankfully after passing me on the exam, he simply said to me "the lesson is to not put anything extra on your slides, because that is where the questions arise." As I mentioned earlier, I have a tendency to exaggerate, and this simple statement changed the way I looked at how I would present research from then on. I am pleased that I got to know him better during our time working for ISSRE, and I am sincerely glad he agreed to be on my committee. I got to know Dr. Tom Honeycutt through his software engineering course, where, I can safely say, he portrayed the field in ways I had never considered. Always a man with a smile and a pipe, I am thankful for his insights into my work and his interest into my progress over these five years. Dr. Jason Osborne was introduced to me through the research of one of my colleagues. Ever since then, his guidance in statistical methods and his knowledge of software reliability has been a boon for everyone in our research group.

Speaking of our research group, I often heard second hand around the department about "Laurie's students" and what we were up to. I am extremely proud to say that I am a member of the Realsearch Group along with some of the best people I have met at NC State. The thing that makes me most proud about our group is that, unlike a lot of graduate students, we are involved in our department. From open houses, to conferences, to managing labs, to being teaching assistants – it makes me proud to know that I work with people who are not only good researchers, but also excel at teaching and service, which is important for any academic. If I went into the specific reasons why I respect each of these people, I dare say it may be longer than the dissertation itself. Thanks to Nachi Nagappan, the first student mentor I had. Thanks to Lucas Layman and Sarah Smith Heckman, the other two corners of what I thought of as "the trio" with me, as we collectively served as Laurie's right hand in teaching and whatever else that needed to be done. Thanks to Dright Ho, Jiang Zheng, Yonghee Shin, Michael Gegick, Hema Srikanth, Andy Meneely, Mei Nagappan, Stephen Thomas, Paul Otto, Laurie Jones, and Colin Butler. My sincere thanks to the staff at NC State for all that they do for the department; without them, we students most certainly would not graduate.

The love and support of family needed to complete this journey can never be exaggerated. Thanks to my father, Steve, who taught me my first lessons about technology and research. Thanks to my mother, Nancy, whose support is never-ending, even when her knowledge of technology didn't allow her to follow along any more. Thanks to my siblings, Beth and Kevin, who balance my geekness quite effectively in the family. Thanks to my in-laws, Dan and Linda Jones, who accepted me into their family without the slightest hesitation.

Saved for last is of course my wife, Amanda, without whom I would be lost. She is by far the smartest woman I know, who could have gone into any field she set her mind to, and she chose to teach young children about reading. The sacrifices I see her make day after day for her kids are astonishing and come with an amazing amount of love. She is my greatest inspiration, my best friend, and deserves every bit of praise I can heap upon her.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Software development managers balance lifecycle costs during development and maintenance with potential profits from sales [21]. Releasing a software system early may gain an initial boost in profits by being first to market, but profit and customer confidence could easily disappear if field failures are found due to a lack of quality. Thus, researchers in the field of software reliability engineering have developed numerous techniques that can direct developers to areas of the code base that contain faults[1] [22, 23, 28-33, 35-37, 43-45, 47-50]. However, many of these techniques require computationally and resource intensive aspects, such as large call graphs, that can take hours to run and several gigabytes of disk space [9]. Based upon how the system is being used in the field and the relative severity of some of these potential faults, delaying product release to address these potential faults may allow competitors to enter the market earlier and gain a larger market share, thus eliminating some of the cost-benefit of performing some of the reliability techniques. Software development managers could benefit from information that can be provided quickly and efficiently during the development process to provide guidance in determining when software is "good enough" to release.

[1] The IEEE defines a fault as "an incorrect step, process, or data definition in a computer program."

Some insight into when to release a product may be gathered from the information generated during the development process itself. During development, programming teams will produce a variety of *software development artifacts*. A *software development artifact* is an intermediate or final product that is the result or by-product of software development [24, 25]. Some software development artifacts are created by the development team, such as source code and design documents. However, other development artifacts, such as change records in a source control system, defect records, and test case logs, are generated throughout the process and are used for support purposes [1].

For example, change records might be referenced to track development progress or to learn the current version of a file in the system. However, these development artifacts can also provide information about underlying relationships within a system that would normally not be apparent [4, 6, 14, 27, 52-55, 61, 63]. Change records can show how components interact with one another or how a system is evolving during development by examining what areas of the system change together and where new *tracks* are appearing [4, 6]. A *track* is a grouping of files that have been changed together to address a defect. Managers can use detailed change information to monitor progress throughout development as sections of code enter and exit test. Testers could use information about how components of a system tend to change together as well, since they could use this knowledge to direct their testing efforts whenever a new track is created.

Several techniques used during development incorporate semantic data, such as techniques based on call graphs showing exactly which files have been affected. Our technique focuses on the use of historical data as opposed to semantic or dynamic data.

There are three main advantages of using historical, empirical data as opposed to semantic data when directing development efforts. First, the cost in system resources can be exceedingly high using a semantic technique. Research by Breech et al. in comparing dynamic impact analysis techniques showed that from a selection of eight programs, seven programs required either more disk space than the test machine had free (15GB) or took over two hours to run [9]. The second advantage is that historical data can show where previous maintenance has taken place or how the system is being used in the field. With this actual operational profile information, development and maintenance efforts can be prioritized to address areas of the system that are under more use. Finally, using software development artifacts allows analysis techniques to treat all files in a system the same, whether those files are source code, media files, documentation, or any other file type. Dynamic fault analyses may only be limited to source code of one particular language; however, faults that appear because of non-executable files can be just as problematic in a software system [20, 55].

Software development artifacts can be used to identify *association clusters* in a software system that would not normally be apparent. An *association cluster* consists of a set of development artifacts, such as files, packages, and blocks of code, that exhibit a specific relationship with the other development artifacts in the cluster [6]. For example, association clusters generated from file change records created from defect removal efforts will consist of files that, over a set of defects, tended to have to be modified together. If a number of files tend to change together to fix a defect in a particular feature of the system, then those files would surface as an association cluster. Similarly, association clusters created from test case records would provide information about what areas of the code are often being tested together. These association clusters could correspond to functional requirements or execution paths, thus exhibiting underlying relationships of the software system that might not readily be apparent. Earlier research has shown that software development artifacts can be used in software change analysis to discover underlying relationships within a code base [4, 6, 14, 27, 52-55, 61, 63]. These relationships have in previous research been used to represent design decisions and for program comprehension. Our work adds to this body of knowledge by introducing a new technique for illuminating these relationships and then leveraging them to help guide development and maintenance decisions.

In empirical software engineering, theory should be integrated into research to advance the body of knowledge [15]. Through this research, an explanation type theory will be built and evaluated. An *explanation type theory*, also called a "theory for understanding," is used to provide understanding of a phenomenon, however its goal is not to predict precise results. The main purpose of theories of this type is to guide others as to how the world may be viewed under certain circumstances [15].

The explanation type theory being built and evaluated through this research is stated as follows:

Decision support for software development decisions is more resource efficient with respect to time, memory, and developer overhead when based upon relationships between software artifacts illuminated via a singular value decomposition of actual usage information than when based upon semantic techniques.

The core motivation behind our theory and this research is providing decision support for industry in instances where resource efficiency and overall time are critical, such as when a product needs to be released in the confines of a strict schedule. As has been discussed, there are numerous techniques available that can, with enough time and computational resources, can provide effective and safe impact analysis and regression test selection. However, there are also instances in industry when time and resource constraints limit the amount of effort that can be put forth into these techniques. We want to address this issue by adding to the body of software engineering knowledge regarding techniques that can provide effective decision support with less overall resources for non-critical software systems, thus helping teams to make software quality decisions that ordinarily could not. There is a trade off of inclusiveness and precision with the resources required to execute the technique, and we are exploring the latter option in this work.

The main contribution of this research is a process that helps guide developers through the use of singular value decomposition (SVD) and software development artifacts to provide insight into various software development decisions. The decision areas include impact analysis, regression test prioritization, and the filtering of static analysis results.

The framework proposed in this research is called Software Development Artifact Analysis (SDAA). SDAA provides guidelines for selecting and mining software development artifacts, generating association clusters, and then leveraging those clusters within the development process to guide development decisions. Artifacts can be gathered from various tools within the development process, such as change management systems, source control systems, and defect tracking systems. The data from these artifacts are compiled into a matrix that relates a particular code unit to other code units with regards to that artifact. For example, if a small system consisted of five files, the matrix would be 5x5 and the values in the matrix would correspond to metrics about a particular development artifact, such as the number of times two files have changed together. A singular value decomposition (SVD) [13] is performed on the matrix to generate the association clusters. The results of the SVD can then be utilized to guide software development decisions.

In this work, we will describe the steps of the SDAA framework and how they are used to help create analysis techniques based upon development artifacts and SVD. We will specifically discuss three techniques that we created using SDAA, including impact analysis, regression test prioritization, and static analysis filtering. To validate our techniques, we performed industrial case studies with each with the cooperation of IBM. Further, we also evaluated our techniques with six different open source projects to test how they would work outside of an industrial setting. The background and related work for each technique will be provided in the corresponding chapter.

Chapter 2 discusses general background research relevant to our efforts, with more specific related work found with related subsequent chapters. In Chapter 3, details will be provided on the steps of the SDAA framework. Chapters 4, 5, and 6 outline three techniques created using SDAA to guide development decisions. Impact analysis, regression test prioritization, and static analysis signatures are discussed in Chapters 4, 5, and 6, respectively. Each chapter describes background information, how the technique was formed, our experiences with an industrial project, further experiments with open source projects, and how our techniques compare to currently available techniques. In Chapter 7, we discuss our contributions to the field through this research and future work.

CHAPTER 2

BACKGROUND

In this chapter, we will discuss the general background to the SDAA framework. In particular, we will examine the use of singular value decomposition in software engineering research and other uses of software development artifacts.

2.1 SINGULAR VALUE DECOMPOSITION

Singular value decomposition (SVD) is a linear algebra technique that decomposes a given matrix into three component matrices [13]: (1) the left singular vectors; (2) a set of singular values; and (3) and right singular vectors. The two matrices that are made up of singular vectors provide information about the structure of the original matrix. The singular values describe the strength of the given components of the original matrix. The SVD theorem [13] states that given a matrix M, then there exists a decomposition of M such that $M = USV^T$.

The SVD of a matrix can also be described geometrically. The SVD shows that the values of any matrix *M* can be reconstructed by a rotation (*U*), followed by increasing the matrix values (*S*), followed by another rotation (*V*) [60]. For example, if *M* represented coordinates that generated a three-dimensional shape, then that shape could be constructed from the rotational information in U and V, along with stretching the shape out to its proper size with the information in S [60]. This type of decomposition can be important and useful in that the rotational matrices isolate the key components of the original matrix, finding relationships between the various data points, while the strength matrix indicates which of the key components illuminated in the rotational matrices are the most important [13, 60]. In our research, this core idea of isolating key components of the original matrix is the basis for using the SVD. When the matrix is comprised of change records, fault information, or some other data from the development process, these key components highlight underlying structures in the code base.

The SVD has other uses in computing. For instance, this technique can be used in image and signal compression. A gray scale image could be represented as a two-dimensional matrix made up of intensity values, indicating the darkness of a particular pixel. In this instance, we could treat the image matrix itself as the original M matrix and perform a SVD on it. Once the decomposition is completed, the resulting matrices, USV^T, can also be represented as the sum of component matrices, as shown in Equation 2.1:

$$M = u_1 s_{1,1} v_1^{\,T} + u_2 s_{2,2} v_2^{\,T} + \ldots + u_k s_{k,k} v_k^{\,T} \qquad \textbf{(2.1)}$$

where k is less than or equal to the rank of matrix M. This provides a rank-k approximation of the matrix. The first component of this factorization indicates the component that has the largest singular value and contributes the most variability to the overall matrix. As subsequent components are added together, the matrix, and thus the image itself, begins to re-take its original form [13].

Osinski et al. created a clustering algorithm based upon SVD to improve search queries on a set of documents [38]. They built an original matrix based upon keywords in the document set. The SVD was performed on this matrix to generate clusters of documents that were similar based on their keywords. Enough clusters were gathered to account for 90% of the variability of the original matrix, with the remaining clusters discarded as signal noise [38]. The documents were then assigned to clusters based upon which cluster they had the closest association with. Anecdotal evidence from users who were presented with the clusters generated with this study found that 70-80% of the clusters were useful and over 75% of identified cluster labels were correct [38].

2.2 USING SOFTWARE DEVELOPMENT ARTIFACTS

Other research has been performed in leveraging the benefits of development artifacts in various aspects of the software process. Ball et al. performed a clustering analysis based on sets of change records to the P5CC compiler that divided the system into five specific functional groupings [4]. Each of these functional groupings mapped directly to a particular functional requirement for the compiler, including abstract syntax tree (AST) creation and code generation. Their research showed that data could be gathered from sets of change records to increase program comprehension and that impact analysis could be performed using this information [4]. Ball's clustering algorithm used a probability measure to determine if a file is likely to change with another file. Our technique provides a similar measure of likelihood as represented by the singular value of the cluster.

Further work expanded on the idea of mining change records to isolate clusters of files within a software system to drive program comprehension. Beyer and Noack's work expanded on Ball's research by creating clusters of files using a co-change graph to plot files onto a two- or three-dimensional graph using an energy-based graph layout [6]. Files that contained more edges between them were closer together on the graph, thus creating clusters of files. These clusters of files could be directly identified and related to functional requirements or third party components within the system [6]. Other research by Gall also generated association clusters of files within a system for program comprehension [14]. His method used a set of commonalities that could be detected from change logs (i.e. files that were edited on the same day by the same person) to create his sets of sub-modules.

CHAPTER 3

SOFTWARE DEVELOPMENT ARTIFACT ANALYSIS (SDAA)

SDAA is a framework that provides developers the means to step through the application of SVD analysis with software development artifacts to help guide development decisions. The specific development decisions can vary between projects and teams, depending on what artifacts are available and what the team is interested in examining. In this research, we highlight three specific techniques derived from SDAA to guide development decisions: impact analysis, regression test prioritization, and filtering static analysis alerts. The possible development decisions are not limited to these three, and we will describe others in Section 3.1.

As mentioned, the two central components to SDAA are software development artifacts and the SVD. By performing a SVD on a matrix composed of software development artifact data, association clusters are created. These association clusters portray an underlying structure or relationship in the system based upon the type of development artifact that is used. The association clusters allow us to gain a better understanding of how artifacts of the system are interconnected. In this section, we will describe the process of SDAA, which includes determining the type of artifact to be used, deriving the association clusters, and interpreting the results of the analysis. Figure 3.1 outlines the steps of the SDAA. Each of these will be described in the subsequent subsections.

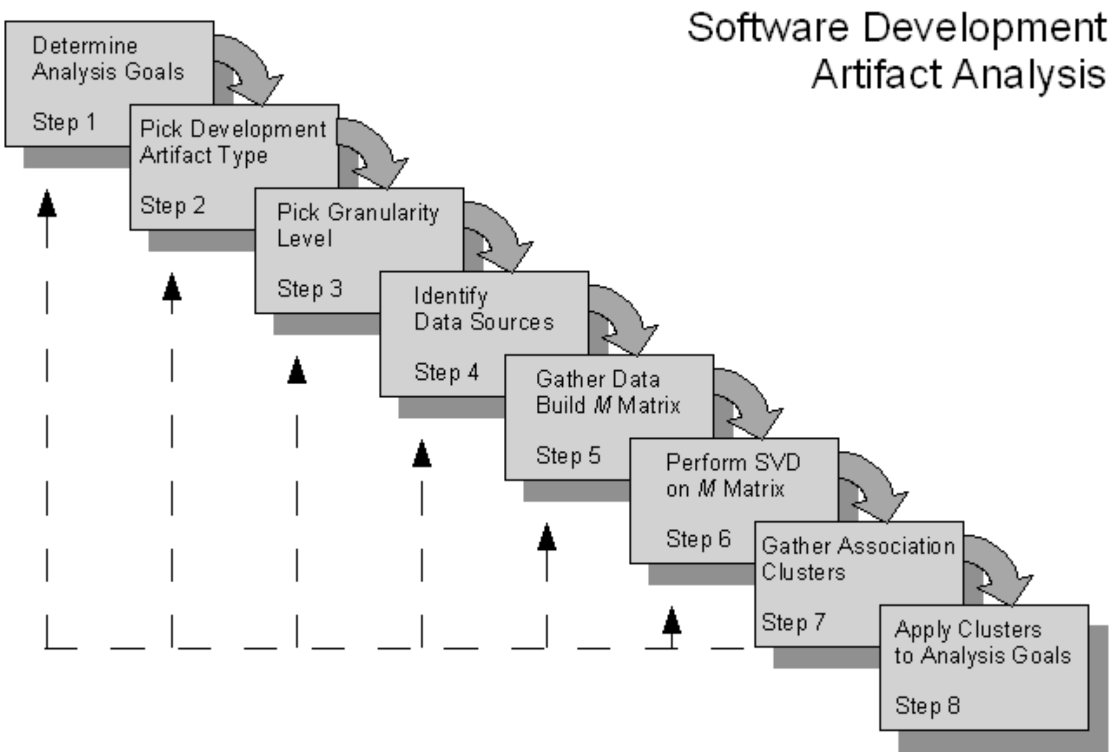


Figure 3.1: SDAA steps

3.1 STEP 1: DETERMINE ANALYSIS GOALS

The analyst must first determine the goal of the analysis to decide whether a particular set of artifacts will illuminate the desired underlying structures. Table 3.1 shows some example software development analysis goals, the artifacts that can be used for that type of analysis, and from where they are gathered. The nature of the development artifact dictates the type of analysis that can be performed.

Table 3.1: Types of software development artifacts

Goals	Development Artifact	Gathered From
To determine the consequences of a set of changes; To prioritize system test cases	Change records derived from test	Source control system, defect tracking system
To determine the effectiveness of test coverage	Change records derived from field failures	Source control system, defect tracking system
To perform a root cause analysis	Defect records	Defect tracking system
To determine the usability of a system	Field failure records	Defect tracking system
To prioritize system test cases; To visualize execution paths through a system	Testing records	Test tracking system, defect tracking system

3.2 STEP 2: PICK DEVELOPMENT ARTIFACT TYPE

Artifacts are created throughout the development process. Table 3.1 provides some example artifacts and artifact sources based upon some potential analysis goals. Some artifacts are generated as a "side effect" of regular software development. For instance, as developers write and manage code in a source control system, the system may generate change logs that record how files have changed. Other artifacts could be metrics related to the other artifacts, such as lines of code or methods per class. Almost any form of meta-information regarding the software being created could be considered a development artifact.

Artifacts are also generated during the testing and maintenance phases of development. Some more complex source control systems are integrated with a defect tracking system. With these integrated systems, changes can be associated directly with the defect record that the changes are addressing, providing detail about how changes are linked together. Information from defect tracking systems allows us to isolate changes to those made under specific circumstances. For example, changes derived from defects found during system test could be compared to changes derived from field failures discovered by customers.

Defect records found in defect tracking systems can also provide valuable information. Defect records could be gathered to find associations based on particular classes or types of defects, such as null pointer dereferences. If specific defect classification information is not present, then the natural language text of the defect descriptions could be used to make further associations. Field failures could also be examined in a similar fashion, providing information about what areas of the system users are having difficulties with. Musa created similar clusters of failure sets to aid in the description and tracking of software defects [29].

The development artifact that is selected should have an inherent connection to the goal of the desired analysis. Analysis goals that involve end users should utilize field failure artifacts, customer feedback, or other information gathered after the system has been released. Testing records and change records derived from test are appropriate artifacts for goals that are directed towards effectiveness of testing and test coverage. Change records grouped by the defect or failure type that motivatse the changes provide association clusters that can address goals regarding system execution paths and impact analysis.

3.3 STEP 3: PICK GRANULARITY LEVEL

Once the type of artifact has been determined, the granularity level of the artifact will determine what type of code unit must be gathered for the analysis. We define the granularity level of an artifact to be the level of abstraction in the software system at which the analysis will be performed, such as line of code, method, class, file, or package. The granularity level should be appropriate for the analysis goal as each different type of code unit will portray a distinct view of the functional modular structure of the system. For instance, if directories or Java packages are selected as the code unit granularity, then the analysis will specifically target connections between these higher level components. If files are chosen as the code unit, then the analysis will still pick up some of the logical component interconnections, but will also provide information about which files are most important within its own component.

The main limiting factor on the code unit to be used in the analysis is the availability of development artifacts and the granularity level at which they are recorded. Source control systems often record when and how files change, but it may be difficult to gather information about how individual functions or methods change. Similarly, testing records might have information down to the line of code level, but this information is often in a constant state of flux, making any associations between particular lines of code difficult to use over the course of a system's lifetime as code is added and deleted.

3.4 STEP 4: IDENTIFY DATA SOURCES

Source control systems are a valuable source for gathering development artifacts. When a developer checks a file in to a source control system, the system typically records the time of the check-in along with information about the developer and the nature of the change. Individual changes can be often be linked together into *tracks*, either through a specific mechanism in the source control system that records that information or by examining change record check-in information. A *track* is a grouping of files that have been changed together to address a defect. With information regarding tracks, we are able to ascertain how files, directories, packages, or other code units change together.

3.5 STEP 5: GATHER DATA AND BUILD *M* MATRIX

After the granularity and the artifacts for the analysis have been identified, an analysis matrix should be generated that contains the software development artifact code units along each axis. If the analysis is looking for relationships among a single type of artifacts, such as files that change together or static analysis alerts that tend to appear together, then the *M* matrix will be a symmetric matrix, as shown in Equation 3.1.

$$M = \begin{bmatrix} x1 & y1 & z1 & \dots & \dots \\ y1 & x2 & y2 & z2 & \dots \\ z1 & y2 & x3 & y3 & z3 \\ \dots & z2 & y3 & x4 & y4 \\ \dots & \dots & z3 & y4 & x5 \end{bmatrix} \qquad \textbf{(3.1)}$$

However, a non-symmetric matrix is perfectly acceptable if the analysis is looking for a relationship between two different development artifact types, such as what static analysis alert types appear in post-release field failures.

The values within the analysis matrix show how the code units interact through the development artifact and determine how the association clusters will be interpreted. For example, if the values in the matrix indicate the number of times that two files have changed together in response to a failure, then the association clusters will represent failure paths found in the system and files that are likely to be linked in some way. However, if the data set is limited to only field failures, then the analysis is focused more on user operational profiles and what areas of the system users are executing and discovering system failures. Further, if we are using two types of artifacts in a given analysis, we place the development artifact of most interest on the rows of M, which precipitates a need to use U for gathering association clusters because the row artifact designations will stay the same, while the columns become the association cluster identifiers. For example, if we want to see how static analysis alerts are related through files and what alert types tend to appear together in files, we would use alert types as the rows of M and files as the columns.

3.6 STEP 6: PERFORM SVD ON *M* MATRIX

To determine the strength of these associations and to generate the association clusters, we perform a SVD of this matrix. An example SVD of a matrix M provides three matrices, shown in Equations 3.2, 3.3, and 3.4:

$$
M = \begin{array}{c} \\ F1 \\ F2 \\ F3 \\ F4 \\ F5 \end{array}
\begin{array}{c} \begin{array}{ccccc} F1 & F2 & F3 & F4 & F5 \end{array} \\
\begin{bmatrix} 25 & 10 & 0 & 0 & 0 \\ 10 & 31 & 21 & 0 & 0 \\ 0 & 21 & 24 & 0 & 0 \\ 0 & 0 & 0 & 15 & 12 \\ 0 & 0 & 0 & 12 & 17 \end{bmatrix} \end{array}
\tag{3.2}
$$

$$
U = V = \begin{bmatrix} -.29 & 0 & .9 & .31 & 0 \\ -.76 & 0 & -.02 & -.56 & 0 \\ -.59 & 0 & -.43 & .69 & 0 \\ 0 & -.68 & 0 & 0 & -.74 \\ 0 & -.74 & 0 & 0 & .68 \end{bmatrix}
\tag{3.3}
$$

$$
S = \begin{bmatrix} 51.1 & 0 & 0 & 0 & 0 \\ 0 & 28.4 & 0 & 0 & 0 \\ 0 & 0 & 24.8 & 0 & 0 \\ 0 & 0 & 0 & 4.1 & 0 \\ 0 & 0 & 0 & 0 & 3.9 \end{bmatrix}
\tag{3.4}
$$

The U and V matrices provide information as to the structure of the association clusters, while the singular values from the S matrix represent the amount of variability each association cluster contributes to the original analysis matrix. Note that U and V are equal, due to M being a symmetric matrix. The U and V matrices provide an orthonormal basis that we will use generate the association clusters.

Another way to view the SVD of M is by looking at USV^T as a sum of sub-matrices [13, 60]. By taking the k sub-matrices out of n total sub-matrices (where n is the rank of M and also the number of singular values in S), we get the k-rank approximation of M. Equation 3.5 shows the breakdown of the SVD into a sum of sub-matrices.

$$M = USV^T$$
$$M = \sum_{i=1}^{k} u_i s_{i,i} v_i^T \tag{3.5}$$
$$M = u_1 s_{1,1} v_1^{\ T} + u_2 s_{2,2} v_2^{\ T} + u_3 s_{3,3} v_3^{\ T} + u_4 s_{4,4} v_4^{\ T} + u_5 s_{5,5} v_5^{\ T}$$

Each of these sub-matrices represents an association cluster. Notice that each association cluster sub-matrix is made up of one singular value from *S* and from identical vectors from *U* and *V*.

3.7 STEP 7: GATHER ASSOCIATION CLUSTERS

Once the SVD has been performed on an *M* matrix, the association clusters can be gathered from the *U* matrix. If *M* is a symmetric matrix, then *U* and *V* are equivalent. If *M* is asymmetric, then *U* and *V* will contain different values. However, the clusters gathered from *U* often are the same as those from *V*, but with different singular values. Thus, by examining *U*, we will still get a good sense of the interconnections within the system.

In the *U* matrix, the development artifacts appear in the rows of the matrix, while the association clusters appear in the columns. Every artifact type that has a value in an association cluster column, however, does not necessarily become a part of that cluster. A threshold value is used to determine if the "strength" of the interconnection is high enough to warrant inclusion. During the course of this research, we will discuss our evaluation of threshold values.

A cluster's strength, represented by the size of the singular value coupled with it, indicates the amount of variability that the association cluster provides to the original analysis matrix [60]. Dividing a cluster's singular value by the sum of all the singular values provides the percentage of how representative the cluster is of the original matrix. A high singular value indicates that that association cluster is more prominent in the analysis matrix, due to a greater number of changes that have occurred. A high singular value could be indicative of a particularly problematic section of code or a new feature that has just been introduced into the system and is experiencing its first rigorous testing.

Duplicate clusters and clusters that are subsets of other clusters can often appear in this type of analysis. The third association cluster shown in Equation 3.3 is a subset of the first cluster, while the fourth association cluster is comprised the exact same files as the first cluster. All three sets of files contain different values associated with the files. The singular values of these clusters provide some indication as to how they should be analyzed. The first cluster has a singular value of 51.1, representing 45% of the overall variability in the matrix; the third cluster has a singular value of 24.8, representing 22%; and the fourth cluster has a singular value of 4.1, representing 4%. These percentages are found through dividing the singular value by the sum of all the singular values and show that the first cluster defines the majority of the information regarding these files. Clusters three and four are, in effect, the same as the first cluster.

Another way to interpret these percentages is to only analyze clusters that account for a sufficiently large percentage of the original matrix (such as 90-95%). Osinski used 90% as a "threshold value" for accepting clusters as being important to the overall analysis [38]. Using a similar threshold value for our illustrative example, we find that the first three clusters make up 93% of the overall variability of the original matrix, and the other two clusters are considered signal noise and removed from analysis.

3.8 STEP 8: APPLY CLUSTERS TO ANALYSIS GOALS

In this step, association clusters are now available that describe a type of relationship in the system between various system artifacts. How these relationships are interpreted and acted upon is dependent on the overall goal of the analysis and the type of artifacts used. Take, for example, a matrix M that is comprised of static analysis alert types as the rows and files as the columns, and the values of the matrix represent the number of times that a given alert type appears in a file. Then the resultant clusters from the analysis indicate what static analysis alert types appear together. Using this information, developers could begin to look for patterns in their development mistakes and could look for those patterns whenever static analysis tools are used.

3.9 LIMITATIONS

Each technique generated using SDAA will share some common limitations with the use of software development artifacts and SVD. First, the efficacy of the techniques is directly related to the quality of the data sources from which the software development artifacts are gathered. For example, if a development team is using an ineffective static analysis tool, the efficacy of our technique cannot improve upon the overall efficacy of the static analysis tool. Our technique is dependant upon reliable sources of development artifacts to work properly. Second, because of our use of historical data, our technique can only interpret and predict events or data for things that have evidence in this history. For example, our impact analysis technique cannot determine the impact of a brand new module in the system, since there would at that point be no historical evidence as to how that module interacts with the rest of the system.

3.10 SUMMARY

In this chapter, we have outlined the steps in the SDAA framework for using software development artifacts and SVD to help guide development decisions. The eight steps are intended to be a guide as developers and managers seek to incorporate mathematically rigorous methodology for highlighting relationships between different aspects of a software system. Throughout the remainder of this work, we will describe our experiences using SDAA and the techniques born from it. Empirical results to date, as will be discussed, indicate the use of the technique is appropriate for non-critical software, particular products with time and resource constraints.

In our work, nearly all of these steps were automated using simple scripts and Java programs. These scripts gathered the development artifacts from a software development tool, such as a source control system or static analysis program, and put the artifacts into a MySQL database for easier management. We then used Matlab scripting to query the database, perform the SVD, and record the generated associated clusters. Matlab scripts were also used to apply the clusters by comparing the clusters to incoming sets of development artifacts. An example script is provided in Appendix A.

Chapters 4, 5, and 6 will illustrate specific examples of the correlation between analysis goals and the use of clusters. In each chapter, we will use the steps in the SDAA framework to leverage various types of development artifacts to address the goals of impact analysis, regression test prioritization, and static analysis filtering.

CHAPTER 4

IMPACT ANALYSIS USING DEVELOPMENT ARTIFACTS

During development, testing, and maintenance, a modification made to a system can often have side effects. These side effects may not be handled properly because developers might not be aware of all the interconnections in the system and the implication of the modification. Developers thus may not make all the changes necessary to properly enact a system modification. We define a *system modification* as an action taken by a developer on the system to repair a fault or implement a feature change according to a given requirement.

Developers can minimize adverse side effects and prevent fault injection resulting from system modifications through impact analysis techniques [3]. *Impact analysis* is defined as "the determination of potential effects to a subject system resulting from a proposed software change." [3, 7] Current impact analysis research follows two different basic methods in determining the effect of a change: either a static or dynamic analysis of the code base [3]. In a static analysis of the code, an impact analysis methodology could require that the developer build a call graph or other effective diagram of the system to discover the areas of the system that are interconnected. In a dynamic analysis, an impact analysis methodology would require the compilation and possibly execution of the source code, thus determining the effects of a change through actual execution paths. Current impact analysis techniques that utilize call graphs, dynamic executions of the system, or static code analysis mainly focus on changes in the system's source code [10, 18, 37, 41].

End users of a system, however, can find numerous failures that do not originate in source code, but in non-compilable, non-executable files, such as media files, documentation files, and help files [56]. Failures that arise in these non-executable areas can be just as severe as a fault within the source code itself [20]. While some dynamic impact analysis techniques have been shown to be safe[2] under most circumstances [36, 37], these techniques may not be capable of determining the impact of a change that occurs solely in a non-executable file that is never compiled with the source code. Some static impact analysis techniques can take into account these non-executable files, however these techniques do not consider trends in actual system usage or the fault-proneness of the set of files impacted. Without usage trends, the results of semantic impact analysis may require more effort to determine exactly which areas of the system have the highest risk of containing a latent fault [36].

Using SDAA, our goal is *to provide an impact analysis methodology that uses historical change records for both executable and non-executable files in a software system to identify potentially-affected areas of a system modification.* By utilizing change records, we can show what files tend to change together for development purposes or in response to repairing faults in the system. A *change record artifact* provides the documentation for a change made to a single file for the purpose of a system modification. Analyzing how tracks

---

[2] An impact analysis technique is considered *safe* if that technique is guaranteed to find every true positive result.

change can provide an indication as to how files interact with one another to perform a system modification [4, 6]. Our technique's main difference from other impact analysis techniques is that ours is based on historical change data rather than execution and/or code semantics. The benefit we gain using historical change data is that every file stored in change management is analyzed similarly, whether that file contains source code, object code, media, or documentation. By leveraging change records as the software development artifact, SDAA provides an empirical method for determining the impact of a change, placing focus on the areas of the system that have been used in testing and by the customer.

In this chapter, we will describe how SDAA can be leveraged through change records to provide an impact analysis of a proposed change to a software system. We will describe in this chapter related impact analysis work and how these techniques correlate with our impact analysis methodology. We will also discuss an industrial case study and an open source case study on the effectiveness of our impact analysis technique and how our results compare to those of other impact analysis techniques in identifying potentially affected areas of a system. Further, we will discuss how this technique can apply to projects that may not follow rigorous development methodologies, such as open-source projects.

4.1 BACKGROUND

Dynamic impact analysis techniques rely upon information gathered from a system during runtime, often gathered through execution of the system or test suites with an instrumented code base [18, 37]. Orso et al. compare two such dynamic techniques, `CoverageImpact` and `PathImpact`, to determine the major differences in cost and effectiveness. These two techniques examine call graphs and execution records from a fully instrumented execution of the system using a comprehensive test suite. `CoverageImpact` utilizes the coverage information of each system execution with program slicing [58] to determine how components of the system are linked together. `PathImpact` uses similar information to build a directed acyclic graph of the system. Both techniques are considered safe, which means that the techniques will catch all of the impacted areas of the system [58], assuming that the tests are reliable and/or the execution extensively uses the system. If the system has little testing, or if that testing is inadequate, the efficacy of these techniques will be severely impared.

`PathImpact` and `CoverageImpact` require dynamic runtime information to determine the impact of a proposed change. In early experiments, the research teams used testing suites created by the development team to determine the call graphs and coverage information on a given system. Orso et al. performed an investigation where they utilized field execution information instead of test-generated execution information to build their models [36]. In this investigation, they used the operational profile information about the system to further determine the percentage of users that would be affected by a change. During their study, they determined that using actual field information can improve the accuracy of an impact analysis effort as compared to the testing suite execution because actual users of the system utilize different portions of the system than simulated users [36]. Orso observed impact sets 15% to 30% smaller than comparable impact analysis techniques.

Static impact analysis techniques do not involve the execution of the code base. Techniques that can be classified as static impact analysis methods work by analyzing information from the software development lifecycle [18] or the semantics of the source code itself [3, 41, 59, 63]. However, Orso demonstrated that static techniques that are "generally imprecise and tend to overestimate the effect of a change" [36, 37]. Orso and Huang both state that this imprecision, manifested as a large number of false positives (up to 90%), comes from the use of static source code with only assumptions as to how the system is used and executed [18, 36].

Our technique is a static impact analysis technique and addresses concerns expressed regarding static techniques. Using SVD, our technique identifies association clusters of files that help alleviate the concern that static techniques generate a large amount of false positives. These association clusters are generated using historical information regarding how files tend to change together in response to faults and field failures. Thus, the association clusters represent general fault paths in the system under actual use. Further, our technique does not require the source code of the system. Using software change records enables our technique to include non-executable files (such as images, documentation, and configuration files) in our impact analysis.

Research is currently being performed in mining and analyzing data from source control systems to identify core components in a software system for use in impact analysis [4, 6, 14, 27, 61, 63]. Zimmerman et al. [63] have created an Eclipse plug-in that performs an impact analysis with regards to the area that a developer is currently modifying while the developer is in the act of modifying the code. The plug-in mines source revision records and creates a set of tuples that indicates what file was modified, what type of object within that file was modified (e.g. field, method, class, etc.), and the name of the object. The plug-in then converts these sets of tuples into transaction rules, indicating areas of the system that tend to change together. As the plug-in builds a large set of these transaction rules during development, the plug-in can make recommendations to the developer as to possible areas of the system that might need to be modified based upon the revisions they are currently making. With a relatively stable code base, Zimmerman reports that 44% of related files can be predicted. However, for evolving systems, the predictions could not work well since the prediction would have to take into account new functions being added constantly [63]. Our technique is similar in that we are leveraging change records in a like manner, except we use SVD as a clustering algorithm to determine the connections between files as opposed to generating transaction rules.

Ren et al. has also created an Eclipse plug-in to predict the impact of code changes for developers to use in-process through white-box techniques [41]. Their plug-in, called Chianti, works by capturing atomic-level changes in the code base. Through the use of call graphs, dependencies are then calculated between these atomic changes to predict what other areas of the code might be affected by a change. Ren performed two case studies on 100 KLOC (thousands of lines of code) system and found that Chianti was able to reduce the number of regression tests depending on the degree of the change implemented. The primary difference between the impact analysis technique used in Chianti and our technique is that Chianti is based upon semantically-based methods in which all associations are created equal regardless of actual usage. The association clusters created in our technique are based upon historical data and, therefore, might be better for prioritization.

Canfora and Cerulo use the descriptions of faults and change records from developers to determine the effect of a change [10]. Their technique compares similarities in the description of a new change to previous changes to identify possible areas that have been affected. If the description of a new fault matches keywords in previous faults, then those files identified by the previous faults may be affected by this new change. Canfora and Cerulo found that, if fault records used consistent keywords and phrases, a recall rate of nearly 98% was possible, with a precision rate around 85% [10].

Our technique is similar to this prior work in that our technique provides a methodology for identifying these associations between artifacts within a given system and then uses that information to guide developers and testers. In the techniques previously discussed, a file could only be associated with one cluster. However, in actual systems, files and subsystems can be interconnected in several different ways. In our technique, files can be associated with more than one cluster, each with a relative strength. Multiple execution flows through a system could indicate that a particular set of files is related to more than one section of the system.

4.2 DETERMINING THE IMPACT OF A SYSTEM MODIFICATION

SDAA provides a methodology that derives associations using SVD based upon a set of change records from testing and field failures. These association clusters of files portray an underlying structure in the system indicating how files tend to be executed, tested, and changed together. In this section, we will go through the steps of the SDAA with regards to the goal of impact analysis, which includes deriving the association clusters from change records and interpreting the results of the analysis.

4.2.1 SDAA STEP 1: DETERMINE ANALYSIS GOALS

The first step in the SDAA is to determine the goal we are trying to reach though the use of development artifacts. During software development and maintenance, impact analysis techniques are used to determine which parts of a software system could be or have been affected by a given system modification. The main goal of this technique is *to identify impacted files from a system modification in a cost effective manner*. There are already numerous impact analysis techniques and tools available in academia and industry that provide guidance for this goal, such as those discussed in the prior section. Several of these techniques are considered *safe*, that is, they are guaranteed to find all potential effects of a change to the source code.

However, there are other aspects of a system that should be taken into consideration when addressing impact analysis. In analyzing the available safe impact analysis techniques, they appear to only be safe with regards to changes in the source code. Most software systems, however, rely on information that is not stored in the source code and in subsequent configuration files. Further, software systems are not comprised only of the source code that is used to build them. Systems often have external media files, such as images and audio, help files, and documentation files. Changes introduced into these files can often introduce faults into a system that could be as severe as any fault found in the source code itself [20]. Thus, there is a need to determine the impact of a change to a non-source file.

Another aspect is the cost of performing an impact analysis. Using some techniques such as `PathImpact`, large call graphs must be constructed for the entire system and maintained. This cost overhead could make the impact analysis technique inefficient for development teams that only require a localized analysis of the effects of a software change.

6.2.2 SDAA STEP 2: PICK DEVELOPMENT ARTIFACT TYPE

We will use change records to show how files have tended to change together over time. To determine how files tend to change together, we will limit the set of change records to only changes that were made to address a specific discovered fault or failure, whether during development or in the field, excluding changes made and recorded during initial code writing. Since these changes would have been made together for a specific purpose that can be linked back to a particular system requirement, we have a relative level of confidence that these files must be related in some way.

4.2.3 SDAA STEP 3: PICK GRANULARITY LEVEL

For this goal, the file level is the most appropriate granularity level to work with. At the module or component level, information about what has been affected by a change is not detailed enough to really provide a developer with any useful information for isolating faults. The line of code or function level could be extremely beneficial for isolating the exact area of change, however, the cost in maintaining details as to what lines of code change together over time could be too high. Further, comparing this information to the location of faults in the system only compounds the complexity of the problem.

4.2.4 SDAA STEP 4: IDENTIFY DATA SOURCES

Source control systems are the primary source for gathering change records. When a developer checks a file in to a source control system, the system typically records the time of the check-in along with information about the developer and the nature of the change. Individual changes can be often be linked together into tracks, either through a specific mechanism in the source control system that records that information or through the examination of change record check-in information. With information regarding tracks, we are able to ascertain how files change together.

Some more complex source control systems are also integrated with a fault tracking system. With these more complex systems, tracks can be associated directly with the fault record that the changes are addressing, providing detail about how tracks are linked together. Information from fault tracking systems allows us to isolate tracks to those made under specific circumstances. For example, changes derived from faults found during system test could be compared to changes derived from field failures discovered by customers.

4.2.5 SDAA STEP 5: GATHER DATA AND BUILD M MATRIX

After appropriate data sources have been identified, an analysis matrix should be generated that contains the system's files along each axis. The values in the matrix represent the number of times that each file appeared in a track with another file and show how the files are connected through change records. For illustrative purposes on how to build the analysis matrix, we will use a set of sample data to generate an analysis matrix. Table 4.1 shows a small sample of the set of the change records that were used to create our example analysis matrix. This example uses a small system consisting of five files.

Table 4.1: Sample change record information

Test Case ID	Fault ID	Track ID	Files Changed
T1	A1	988	1
T2	A2	989	2, 3
T3	B1	990	4, 5
T3	B2	991	4, 5
T1	B3	992	1, 2
T4	C1	993	2, 3
…	…	…	…

We have built an example analysis matrix *M*, shown below in Equation 4.1 using the data from Table 4.1 and additional change records. Thus, File 2 has appeared in a track 10 times together with File 1, 21 times together with File 3, and 0 times by itself (since ***M*(2,2) = *M*(2,1) + *M*(2,3)**). Similarly, File 3 has changed 21 times with File 2 and 3 times by itself.

$$M = \begin{array}{c} \\ F1 \\ F2 \\ F3 \\ F4 \\ F5 \end{array} \begin{array}{c} \begin{array}{ccccc} F1 & F2 & F3 & F4 & F5 \end{array} \\ \begin{bmatrix} 25 & 10 & 0 & 0 & 0 \\ 10 & 31 & 21 & 0 & 0 \\ 0 & 21 & 24 & 0 & 0 \\ 0 & 0 & 0 & 15 & 12 \\ 0 & 0 & 0 & 12 & 17 \end{bmatrix} \end{array} \qquad \textbf{(4.1)}$$

Upon initial examination of this matrix, we note that Files 4 and 5 change together or by themselves. Based on this, it appears that Files 4 and 5 are strongly linked in isolation from the rest of the system. Similarly, Files 1, 2, and 3 are also linked, with Files 2 and 3 having the strongest bond of the three.

4.2.6 SDAA STEP 6: PERFORM SVD ON MATRIX M

To determine the strength of the associations between files and to generate the association clusters, we perform a SVD of this matrix. The strength of the association is determined by the frequency of time the files changed together. A SVD of M (Equation 4.1) provides the following matrices, shown in Equations 4.2 and 4.3:

$$U = V = \begin{bmatrix} -.29 & 0 & .9 & .31 & 0 \\ -.76 & 0 & -.02 & -.56 & 0 \\ -.59 & 0 & -.43 & .69 & 0 \\ 0 & -.68 & 0 & 0 & -.74 \\ 0 & -.74 & 0 & 0 & .68 \end{bmatrix} \tag{4.2}$$

$$S = \begin{bmatrix} 51.1 & 0 & 0 & 0 & 0 \\ 0 & 28.4 & 0 & 0 & 0 \\ 0 & 0 & 24.8 & 0 & 0 \\ 0 & 0 & 0 & 4.1 & 0 \\ 0 & 0 & 0 & 0 & 3.9 \end{bmatrix} \tag{4.3}$$

The U and V matrices provide information as to the structure of the association clusters. The singular values from the S matrix represent the amount of variability each association cluster contributes to the original analysis matrix. Note that U and V are equal, due to M being a symmetric matrix.

A cluster's strength, represented by the size of the singular value coupled with it, indicates the amount of variability that the association cluster provides to the original analysis matrix [60]. Dividing a cluster's singular value by the sum of all the singular values provides the percentage of how representative the cluster is of the original matrix. For instance, the second cluster in this set represents approximately 25% of the variability (28.4/(51.1+ 28.4+24.8+4.1+3.9)) of the original matrix M. Using change records as the development artifact, a high singular value indicates that that association cluster is more prominent in the analysis matrix due to a greater number of changes that have occurred to that set of files. A high singular value could be indicative of a particularly problematic section of code or a new feature that has just been introduced into the system and is experiencing its first rigorous testing.

4.2.7 SDAA STEP 7: GATHERING THE CLUSTERS OF FILES

The values in the *U* matrix correspond to the composition of each association cluster. In Equation 2, there are five association clusters because the rank of *M* is five. The first column of *U*, representing the structure of the first association cluster, is coupled with the first singular value in *S*, representing the strength of that association cluster. Since it is coupled with the largest singular value, the first association cluster represents the greatest amount of variability in the original analysis matrix and is the most prominent association cluster. From the *U* matrix, we see that the first association cluster is comprised of Files 1, 2, and 3, indicated by the fact that the three files all have values with a similar sign. Values with a similar sign (either positive or negative) indicates that the change vectors are moving in the same direction and are thus related in some way. Further, each of these values has a larger magnitude than .1, the threshold we used in our research. A threshold is used when selecting cluster members so that only files with a strong association to the other files are included in the cluster. This threshold is similar to the threshold that Osinski used in his algorithm [38]. In the third cluster, we see that File 1 is its own cluster that can, at times, change without Files 2 and 3. So, in effect, we get two associations out of the third cluster, one with File 1 by itself and one with Files 2 and 3 together.

Note that the values in each association cluster's column vector represents the degree to which that file is likely to change in that cluster. In this way, each file is weighted within that association cluster as to its degree of participation. For example, the first association cluster is primarily composed of File 2 and File 3 due to their higher values. File 1 is a minor participant in this association cluster. If we reexamine the original analysis matrix *M*, we can see the strong correlation between Files 2 and 3 with a somewhat looser correlation with File 1, since these files only tend to change together and not at all with Files 4 and 5. The association cluster in the second column portrays the next most significant cluster, comprised of Files 4 and 5.

The singular values of these clusters found in the *S* matrix provide some indication as to how they should be analyzed. The first cluster represents 45% of the overall variability in the matrix, which can be determined by dividing the first singular value by the sum of all the singular values. The second cluster represents 25% of the overall variability. Further, the third cluster represents 22% and the fourth represents 4%. These percentages show that the first cluster defines the majority of the information regarding these files. Clusters three and four are, in effect, sub-clusters of the first cluster because they contain a similar set of files. At this step in our technique, the matrix *U* can provide information about the likelihood of a change in an association cluster based upon previous change information.

4.2.8 SDAA STEP 8: APPLYING THE CLUSTERS TO THE ANALYSIS GOAL

Using the *U* and *S* matrices generated from our technique, we can determine the possible impact that a new track might have on the system. We can compare the associations gathered from the *U* matrix with a new track. If all the files in a new track are present in a previously-determined association cluster, we know that there is a strong relationship between the changed files and the other files in the cluster and that these files would be the most likely files to be affected by this change. Further, the magnitude of the corresponding singular value in *S* indicates which cluster has churned more within the data set under examination. If the files in a new track do not all appear in the same cluster, then they represent a new execution path through the system. In this instance, files that are associated with each changed file separately can possibly be affected by this change. Finally, the files may have never changed before within the data set that was used to build the *U* matrix. In this instance, no historical evidence exists as to how these files may affect the system, and a new association cluster is will form to represent this new set of changes in a future analysis.

This technique is similar to the cluster rank algorithm used by Osinski et al. in their SVD-based search term clustering algorithm [38]. Osinski multiplied their document matrix by a modified *U* matrix from the SVD to derive the impact that each search term had on a given document. In this fashion, the values from the result vector were used to assign a document to its closest-matching search term cluster [38].

4.2.9 LIMITATIONS

The association clusters are based upon the change records that may or may not be accurate. For example, the main case study we will present later in this chapter uses data from an IBM system. IBM's documented development process and interviews from developers and managers indicated that files that were changed together are related and are addressing a specific fault. The process in place in IBM helps to minimize opportunistic changes. We define an *opportunistic change* as any change that is made in conjunction with a set of changes that addresses a specific fault or requirement but is actually not associated with that set in any way except for being changed at the same time. An example of this would be a developer that is repairing a fault in the system and then also makes an additional change to the system and then checks all the changed code in together on the same failure report. Excessive opportunistic changes can have an adverse affect on the ability of the SVD to identify related system components because the extra opportunistic changes artificially inflate the strength of the relationship between components when in actuality the link is much weaker.

Another limitation is that our technique is not guaranteed to be safe. If there is no historical data regarding a set of files, our technique cannot provide guidance as to where the effects may be. However, our technique may be more practical for a system with a large percentage of non-executable files. Dynamic techniques might not be able to find the same dependencies that our technique finds among these files since dynamic techniques operate primarily on source files.

Current impact analysis techniques typically calculate the impact of a proposed change at the function or line of code level, while our technique is currently being used at the file level. Using a technique that has a granularity level down to a line of code can be beneficial if a source file is large in size, since a line of code granularity would limit the area that a developer would have to analyze to find the affected code. However, since our technique is also focusing on non-executable parts of the system, the file level is the most effective granularity level for our purposes. In most cases, change information does not exist to the line level of some non-executable files, such as help files. Further, the file level is effectively the only appropriate granularity for media files that are included in a system.

4.3 INDUSTRIAL FEASIBILITY STUDY

We performed a feasibility study on the use of singular value decomposition with change records to identify related system files on a large, industrial software system at IBM in Durham, NC. In this section, we will discuss our methods in conducting this feasibility study and how our results compare to actual system file relationships as determined by system experts.

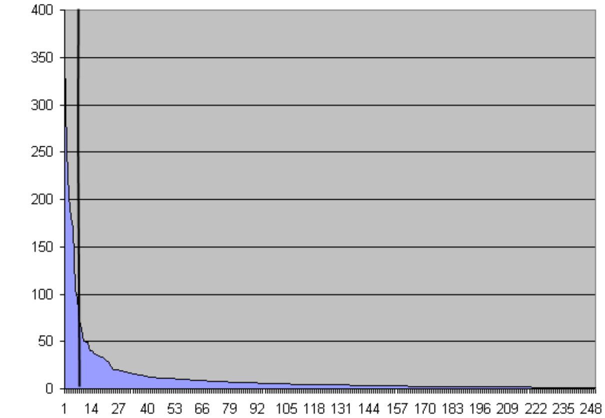
4.3.1 FEASIBILITY STUDY SETUP

We selected Matlab 7.2 R2006a as our SVD tool and began by examining available data sources. IBM's source control and fault management system generates detailed logs on tracks. The project that was selected had three consecutive minor releases. A *minor release* contains small updates to functionality and fault fixes; a *major release* includes a significant change in functionality. Each release of the project contained approximately 21,000 files with over one million lines of code. To protect proprietary information, we cannot provide the quantity of change records analyzed.

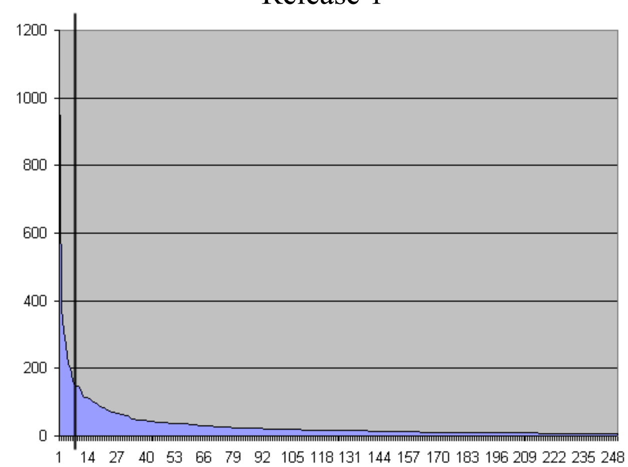
4.3.2 CLUSTER IDENTIFICATION

The goal of the feasibility study was to determine if the association clusters produced by SVD are intuitively identifiable by system experts and thus represent actual system components. Beyer and Noack [6] evaluated their clustering algorithms based upon how accurately the clusters represented sub-systems compared to an actual decomposition of the system. Osinski et al. [38] performed a comparable exercise using multiple sets of documents and interviewing users on whether the search results from the algorithm were useful. We followed a parallel approach to previous research to qualitatively assess the accuracy of the clustering algorithm.

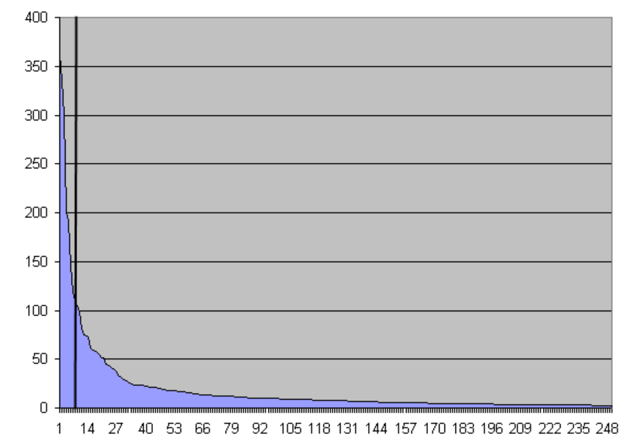
Once we generated the association clusters for the three releases, two authorities on the system, a testing manager and a senior technical staff member, were independently shown the sets of files that comprise the clusters with the highest singular values. The authorities were asked to give each cluster a name based upon the types of files in the cluster (e.g. "system configuration files"). This naming step is not a required step in our technique, however, it aids us in addressing our research goal for this feasibility study. For the purposes of this cluster identification exercise only, we limited the number of clusters for analysis to the first six for each release (18 total), which accounted for approximately 25% of the overall variability in the original matrix. The first six in each release were chosen for this case study because, after the sixth singular value, the singular values drop off significantly and slowly reduce to zero as shown in Figure 4.1. Thus, we selected the most prominent clusters for our identification analysis in this case study. Graphs of the top 250 singular values for each release are shown Figure 4.1. The top 250 are shown because the full graph of all 21,000 singular values is difficult to interpret visually.



Release 1



Release 2



Release 3

Figure 4.1: Graphs of top 250 singular values for the three minor releases

The two authorities on the system independently provided similar names for each cluster. The identifications and singular values for the six association clusters for each minor release are shown in Table 4.2 and are in order by singular value. Further, the main release requirements for all three releases were evidenced in the top six association clusters, indicating that the association clusters can identify system components and order them by velocity of change as indicated by the magnitude of the singular values. Another thing to note about these clusters is that approximately 83% of these clusters include files that are not source code, including license files, help files, configuration files, and images from the graphical user interface. This result indicates the importance of non-executable files in this particular project. From these results, we can surmise that association clusters generated from a SVD using change records can represent specific, feature-related sections of the code base.

Table 4.2: Results from association cluster creation

Assoc. Cluster	**Release 1**	**Release 2**	**Release 3**
1	Patch config files	Release req. 2.1	Release req. 3.1
2	System config files	Database	Release req. 3.2
3	Database	System config files	Graphical user interface
4	Patch information	Release req. 2.2	Source control files
5	Release req. 1.1	License files	Release req. 3.3
6	License files	Graphical user interface	Installation files

4.4 INDUSTRIAL CASE STUDY

From our results from the feasibility study on the potential usefulness of the SVD in isolating files in the system that are related, we determined that there was sufficient cause to continue in this branch of research. From our experiences with the feasibility study, we were able to refine our data gathering and automate our SVD generation. We expanded our research on the same data set to now investigate how an impact analysis technique could be applied using the relationships found between files using the SVD. *Our hypothesis is that a methodology based upon singular value decomposition using historical change records can accurately surface additional files, including non-source files that may be impacted by a set of changes.* In this section, we will discuss the use of our impact analysis technique and how its results compare to those of other techniques in use today.

4.4.1 CASE STUDY SETUP

We continued to use Matlab 7.2 R2006a as our SVD tool and the same data sources as our feasibility study. We first determined the size of the impact sets returned by our technique, and then we investigated the accuracy of those impact sets. We measured the size of the impact sets generated by our technique to determine how much our technique minimized the impact set. A random data splitting technique was used with the three minor releases of an industrial software system in this investigation to create our data sets. We began by randomly selecting two-thirds of the tracks for each release as the "historical data" for our training set, from which we generated a set of association clusters. The remaining one-third of the tracks were then used as our "future set," which would simulate incoming tracks made to perform a system modification. We performed this data splitting exercise ten separate times for each release.

The impact analysis techniques used by Orso et al. [36] and Law and Rothermel [26] are considered safe. As a result, these researchers showed the efficacy of their impact analysis technique by comparing the size of the impact set against that of other impact analysis techniques. A smaller impact set for a safe technique shows that there are fewer false positives while still retaining the full set of true positive results. We utilized a similar methodology to first investigate the relative reduction of the impact set.

We gathered impact sets from the system modification in the future set using three different impact methods: two using our technique (Impact Methods 1 and 2) and another as a baseline (Impact Method 3):

- **Impact Method 1:** Gather all the files that appear in clusters in which all of the newly-changed files appear. For example, if a new track contains files A, C, and Q, a file is considered in the impact set if it appears in a cluster in which A, C, and Q all appear together.
- **Impact Method 2:** Gather all the files that appear in clusters in which any of the newly-changed files appear. For example, if a new track contains files A, C, and Q, a file is considered in the impact set if it appears in any cluster that contains at least one of files A, C, or Q.
- **Impact Method 3:** Gather all the files that have changed in the "historical data" with any of the newly-changed files and does not use any clustering technique. For example, if a new track contains files A, C, and Q, a file is considered part of the impact set if that file has been modified in conjunction with either A, C, or Q in a system modification in the historical data.

4.4.2 CASE STUDY RESULTS

We compared the size of Impact Methods 1 and 2 impact sets to that derived by Impact Method 3. The goal of this comparison is to show that using SVD can narrow the scope of files that should be examined in the event of a system modification to those files that are most strongly connected to the changing files. An example result from Release 3 can be found in Figure 4.2.

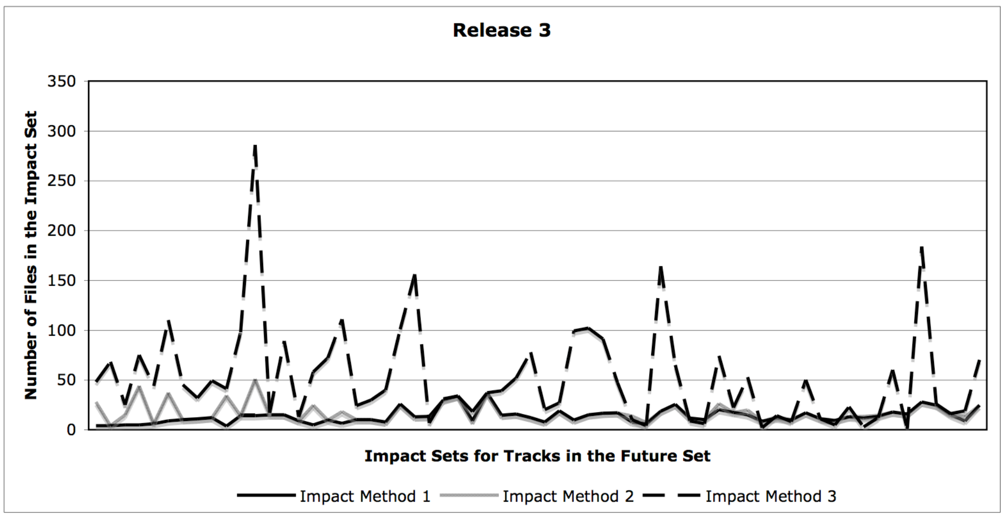


Figure 4.2: Charts of comparisons of impact sets

All three releases followed a similar pattern to the result shown from Release 3 in Figure 4.2. The lines in the chart represent the quantity of impacted files for each track found in the future set. The chart shows that the size of the impact set is generally significantly reduced using either Impact Methods 1 or 2 that utilize the association clusters, on average from 74 files to 20 files with the maximum reduced from 281 to 51. Further, the size of the impact set remained relatively constant, despite spikes in the size of Impact Method 3.

After we evaluated the reduction of the size of the impact set using our technique, we investigated the accuracy of those impact sets. Because our technique generates an impact set based upon historical evidence, the files in the impact set could be considered as a recommendation for further inspection of additional files. Thus, we evaluated the accuracy of the Impact Method 1 by determining whether the files in the impact set appeared in the future set along with one of the files from the new system modification.

For the discussion of our accuracy analysis technique, consider a track in the future set that contains files A, B, and C. Impact Method 1 generates a cluster that contains A, B, C, and D, thereby providing a recommendation that D also be examined. We refer to files A, B, and C as files in the "test track" and to file D as a file in the "impact set."

For our analysis, we considered a file in the impact set a confirmed true positive if that file appeared in any track in the future set with one or of the files from the test track. For example, if D appears in a track in the future set with either A, B, or C, we consider this matching result as a confirmed true positive. However, if D does not appear with A, B, or C in the future set, the conditions of system activity may not have involved these files. Thus, any recommendation that is not a confirmed true positive may either be an unconfirmed true positive or a false positive.

We performed this investigation of Impact Method 1 within each of the three minor releases. We also used the results of Release 1 with the changes from Release 2, and the results of Release 1 and 2 with the changes from Release 3. In an industrial setting, the association clusters adjust as new change records are gathered when new features or faults are discovered. The technique evolves along with the system itself. The results of this investigation can be found in Table 4.3.

Between 21.1% and 55.3% of the files that were indicated as impacted in any given system modification were non-source files and would not have been considered in current semantic impact analysis techniques. The higher number of non-source files impacted partly the result of the type of system under investigation in this case study, but the results do indicate that often non-source files, such as images or help files, can be impacted by a system modification. Also note that the average number of impacted files is in addition to the actual changed files in the system modification.

Table 4.3: Investigation results

Release	Confirmed True Positive Rate	% Non-Source Files	Avg./Med. Track Size	Avg./Med. Size of Impact Set
1	45.4%	35.2%	19.3 / 3	21.1 / 3
2	10.0%	55.3%	40.5 / 4	43.9 / 5
3	39.5%	34.4%	22.4 / 3	21.2 / 3
1 w/ 2	31.5%	21.1%	28.3 / 4	24.4 / 3
1+2 w/ 3	38.4%	35.6%	23.4 / 3	18.1 / 4

Our data indicates that the accuracy of our technique is correlated with the quantity of the number of changes for a set of files under change. For example, in Release 1 and Release 3, there were a small number of components of the system that were under change. The files in these areas had a relatively large number of changes associated with each of them. However, in Release 2, even though there were a larger number of changes overall, the changes were spread out evenly across the system. The SVD creates associations based upon the density of groups of changes across system files, so areas with a higher change density cluster together better than larger areas with a lower density. What this lower change density yields is a larger impact set because the SVD associates more files together, as is the case in Release 2. However, with enough information about how files change together with when the changes from Release 2 were combined with those from Release 1, the confirmed true positive rate improved because the overall change density increased.

In our background research for this work, we did not discover any other empirical analysis of an impact analysis technique that examined the accuracy of the technique based upon future system modifications. Other techniques were validated by examining the size of the impact set, given that the technique was considered safe.

4.5 OPEN SOURCE PROJECT CASE STUDY

In the previous section, we outlined our experiences with an industrial project in which we could be confident that opportunistic changes were kept to a minimum. However, there is a liklihood that some software projects will not follow a similar stringent adherence to software change modification procedures. To evaluate how our technique would perform in a situation where we cannot know whether the number of opportunistic changes have been minimized, we replicated our industrial case study with five open source projects from SourceForge (http://www.sourceforge.net), a popular open source change management and resource site.

4.5.1 CASE STUDY SETUP

The five projects were selected using similar open source project selection criteria as used by Nagappan [31]. On SourceForge, projects are tracked and rated based upon various activity metrics, ranging from the frequency of code modifications to how often people download active projects. Using these metrics, we can gauge whether a project is currently under active development and if the project is considered worthwhile by the community. The selection criteria for inclusion in our study are as follows:

- *Download ranking of 85% or higher.* On SourceForge, projects are rated based upon how often they have been downloaded in comparison to the total number of downloads that occur on the site. If a project has a download ranking of 85% or higher, then that project is in the 85th percentile of projects available from SourceForge and is considered to be a worthwhile and useful project by the community. A project falling into this category would likely be of an acceptable level of quality to be downloaded so frequently.
- *Development stage 4 or higher.* On SourceForge, projects contain a ranking that indicates the project's level of maturity and completion. A project at stage 4 has had at least one beta release, and thus is operational on some level and does provide some form of benefit to its users.
- *Use of Subversion change management software.* SourceForge allows its users to select from two change management software packages, Concurrent Versions System (CVS) or Subversion. While both of these change management systems provide users with mechanisms for tracking software changes through logs, Subversion provides an easily-parsed XML log format which we can leverage to determine which files tend to change together. An example log entry can be found in Figure 4.3.

```
<logentry revision="129">
<author>nenry</author>
<date>2007-07-16T14:50:21.493149Z</date>
<paths>
<path
action="M">/trunk/swt/org/jfree/experimental/swt/SWTGraphi
cs2D.java</path>
<path action="M">/trunk/ChangeLog</path>
<path
action="M">/trunk/swt/org/jfree/experimental/chart/swt/dem
o/SWTTimeSeriesDemo.java</path>
</paths>
<msg>implemented SWT alpha channel.</msg>
</logentry>
```

Figure 4.3: Example Subversion log entry from jfreechart project

As can be seen from the example Figure 4.3, a log entry contains a revision number, the author of the revision, the date of the change, the files affected, the nature of the change (added files, modified file, deleted file), and a message describing the change. The level of detail in the message field varied between the projects and among the project developers, ranging from detailed accounts of a fault fix to no information at all. Using the criteria set above, we selected the following five open source projects:

Table 4.4: Selected open source projects

Project	Type of software	# Files	# Revisions
Abbot[3]	Java GUI testing framework	6282	2460
Grinder[4]	Java load testing framework	3260	3520
HTTPUnit[5]	Web site unit testing	697	779
jFreeChart[6]	Java graphical/charting library	1245	135
StatSVN[7]	Gathers Subversion metrics	595	342

[3] http://abbot.sourceforge.net/
[4] http://grinder.sourceforge.net/
[5] http://httpunit.sourceforge.net/
[6] http://www.jfree.org/jfreechart/
[7] http://www.statsvn.org/

From examining log messages from the five open source projects, it appears that some developers did attempt to avoid checking in any opportunistic changes. However, this assertion cannot be proven across any of the projects since the various developers of the projects changed over time and each had a varying impact on the project. Thus, for the purposes of this investigation, we must assume that opportunistic changes do occur in the projects. The advantage of using SVD even in projects where opportunistic changes occur is that over a large set of changes, files that consistently change together even with opportunistic changes will still cluster together. However, a larger change set is needed when opportunistic changes are present to effectively identify these clusters. When there are no opportunistic changes, fewer change records are required because the signal to noise ratio in the data set is much lower.

We continued to use Matlab 7.2 R2006a as our SVD tool. Using the same three impact methods from the industrial case study, we measured the size of the impact sets generated by our technique to determine how much our technique minimized the impact set. These impact methods were applied in the same manner as the industrial case study, using two-thirds of the changes to created a "historical set" and the remaining one-third of the changes as the "future set." We performed this data splitting exercise ten separate times for each open source project.

4.5.2 CASE STUDY RESULTS

Using a similar method to our industrial case study, we applied the three impact methods to each of the ten random data splits for each of the five open source projects. We first examined how the size of the impact set was affected by our technique. The results of the three impact methods are shown in Table 4.5.

Table 4.5: Impact method size summary

Project	Method 1 Avg. / Med.	Method 2 Avg. / Med.	Method 3 Avg. / Med.
Abbot	3.5 / 3	5.5 / 4	14.3 / 13
Grinder	2.6 / 3	3.1 / 3	12.2 / 10
HTTPU.	4.4 / 4	5.1 / 6	10.6 / 8
jFreeC.	9.4 / 5	15.1 / 11	30.4 / 31
StatSVN	8.5 / 5	10.4 / 8	14.4 / 13

We observed a similar decrease in impact size when we looked at files identified as being related to changed files through the association clusters. This result follows a similar pattern to that of our industrial case study. We then examined the efficacy of Impact Method 1 by investigating its true positive rate. The results of this examination are shown in Table 4.6.

Table 4.6: Investigation summary

Project	Confirmed True Positive Rate	% Non-Source Files	Avg. Revision Size	Avg. Size of Impact Set
Abbot	52.4%	3.2%	7.25	3.5
Grinder	43.2%	2.2%	3.74	2.6
HTTPU.	41.1%	5.9%	5.2	4.4
jFreeC.	19.5%	0.5%	14.8	9.4
StatSVN	35.5%	5.3%	5.4	8.5

On average, the true positive results from our open source study are better than those from our industrial case study, which is contrary to what we hypothesized would happen in an environment where opportunistic change were not regulated in the same fashion as in an industrial setting. Note that the efficacy of our technique is correlated with both the average revision size and the number of overall revisions. With a smaller number of overall revisions (135) and larger average revision size (14.8), jFreeChart had the least success with our technique. We believe that this particular project is a good example of a project in which there are more opportunistic changes. The larger average revision size along with fewer total revisions could be an indication of a lot of development taking place before the developer checks in their work Subversion. Note that in these instances the developer implement more than one feature change to the system before, given that there are fewer instances of source code revisions and the higher number of files revised at a given time. Thus, the signal to noise ratio, where the signal is related files changing together and noise is unrelated files being checked in together, is much higher than that of the other open source projects we examined.

At the other end of the spectrum is the project Abbot. This project had a much smaller average revision size (usually between two to five files) and had a factor of ten more revisions than jFreeChart. We hypothesize that in this project, developers made smaller, directed revisions to the code and checked in the changes immediately, thus reducing the number of opportunistic changes without the need for a rigorous process. The idea of making smaller changes and immediately checking in the code to the source repository is related to several of the tenets of Extreme Programming [5]. The tenets of collective code ownership, continuous integration, and refactoring all would contribute to this phenomenon of numerous, targeted revisions. From examining the results of our investigation, we could infer that any development methodology or project that supports small, targeted revisions to the code as opposed to larger ones would receive the most benefit from our technique.

Through such a methodology, opportunistic changes could be minimized regardless of the commercial nature of the project.

4.6 COMPARISON TO OTHER TECHNIQUES

As has been discussed, we believe the advantage to using our technique is three-fold. First, we contend that our technique requires fewer system resources and less time to execute than other techniques. If a development team is trying to balance speed with effectiveness, our technique may be more advantageous. Another advantage that has an effect on speed and directed effort is the use of historical data. Using historical data, our technique identifies which impacted files have been under the most change or reported as problematic from the field. Weyuker suggests that most software faults are localized within a system[39], and historical evidence would help identify these areas of code easily. Finally, by using empirical change records as opposed to semantic information, our technique can determine the impact that a non-source, non-executable file can have on the project. To determine the scope of these advantages, we examined how our technique compared to both `CoverageImpact` and `PathImpact`, two academically-recognized, safe impact analysis techniques.

During this investigation, we compare the results from two sets of experiments run by Orso et al. and by Breech et al. on the performance of `CoverageImpact` and `PathImpact`, along with the results of our technique using the a large open source project, `gcc`. Further, we compare the time and resources required for each technique.

We used the algorithms provided by Rothermel and Orso in the recreation of `CoverageImpact` and `PathImpact` for our study. We used JAnalyzer[8] as our tool for dynamic instrumentation and generating call paths. We continue to use the same version of Matlab from our previous studies. Further, we followed some of the methodology provided in Orso and Breech as to how to compare impact analysis techniques [9, 37].

4.6.1 ALGORITHMIC COMPLEXITY

The first advantage that our technique has over other impact analysis techniques is in the cost of system resources and the amount of time needed to perform the analysis. A key component in this analysis is in how complex the algorithm is and how the algorithm scales to larger programs. For this section, we define the following variables:

- f is the number of functions affected by a new change set introduced into the system;
- F is the number of files in a given system;
- T is the size of the call trace generated by the f functions;
- R is the number of revisions recorded in the change repository;
- A is the average number of files affected in a revision;
- C is the number of association clusters generated by the SVD; and,
- L is the number of lines of code in the system.

`PathImpact` operates by first instrumenting the code base and executing the entire system using a test suite. A call trace is gathered from the instrumented system during execution, and can be compressed using the SEQUITUR algorithm [37] so that the data is not

[8] http://www.janalyzer.com

too large to store on a single machine. Running SEQUITUR is not always necessary, however, the size of the call graph generated from the call trace can be so large as to exceed system memory [9]. Once system execution and SEQUITUR have finished, `PathImpact` determines the impact of a changed function by looking forward and backward through the DAG. Examining the DAG is proportional to T to examine the impact of a single function change. Thus, if multiple functions are affected by a given new revision, then the total complexity for a full change set is $O(f*T)$.

`CoverageImpact` also begins by instrumenting the code base and running the entire testing suite. `CoverageImpact` uses a single bit vector to record how the functions are exercised during the test run. Static program slicing with a control flow graph is then used to determine the impact of the change of a single function. Since that function could possibly travel through the entire program, in the worst case the complexity for this step is $O(L)$. Thus, as with `PathImpact`, if a new revision contains more than one affected function, the total complexity of this step is $O(f*L)$.

Our technique begins with the gathering of change records from a source repository. The time it takes to gather these records is proportional to the amount of activity the repository has had over time, or $O(R*A)$. Once the records are gathered and placed in the matrix M as described in earlier sections of this chapter, a sparse SVD is performed on the matrix. A sparse SVD is performed because most files in the system will not change with every other file in the system, thus making M a sparse matrix. The complexity of a sparse SVD is $O(F \log F)$. After the SVD has been completed, the impact of the new revision is done by comparing the changed files to the cluster set, which is $O(F*C)$.

The three algorithms' steps, algorithmic complexities, and storage usages are shown in Tables 4.7 and 4.8. Looking at `PathImpact` and `CoverageImpact`, the main time and validity issues arise from the use of the test suite. `PathImpact` and `CoverageImpact` are both reliant on comprehensive testing suites that exercise the entire code base. If there is no automated way to run the system to gather either a call trace or coverage data, the effectiveness of `PathImpact` and `CoverageImpact` decreases. Assuming that a system does have a comprehensive test suite, running that test suite could take a great deal of time, depending on the nature of the tests and the software itself. For example, some of the tests from the industrial project we examined earlier in this chapter contained tests that required the start up and shut down of server applications manually. These particular tests could take up to a full person day to execute. Having any test suite that requires excessive manual intervention increases the overall cost of using the technique.

Depending on the scope of the software system, the system could have a comprehensive test suite that is not overly expensive to run to gather execution data. Even with a good test suite, the building of a call trace for `PathImpact` has been shown to be infeasible in some circumstances as it does not scale well to larger programs. Breech performed a study comparing `PathImpact` and `CoverageImpact` with his own impact analysis technique and showed that from a set of eight systems selected for experimentation [9]. Five of the systems selected could not use `PathImpact` effectively due to resource constraints that stemmed from the size of the system. The overhead for gathering and storage requirements for the call trace resulted in either excessive time allotted for the experiment (two hours) or data sets that exceeded the remaining space on the machine (15GB).

Table 4.7: Impact analysis algorithm steps

	Pre-Processing	**Impact of Change Set**	**Post-Processing**
`PathImpact`	1. Instrument code 2. Run entire test suite 3. Gather call trace 4. Compress using SEQUITUR	For each function that changes, follow the DAG forward and backwards	If test cases change or are added, reinstrument code base and run entire test suite again. The entire suite must be executed so that a full call graph is generated.
`CoverageImpact`	1. Instrument code 2. Run entire test suite 3. Gather coverage information	For each function that changes, perform static slicing and compare	If test cases change or are added, reinstrument code base and run entire test suite again. The entire suite must be executed so that a full call graph is generated.
SDAA Technique	1. Gather change records 2. Perform SVD	Compare all changed files simultaneously with cluster set	After a significant number of revisions (several days of revisions for a mature project, one day for early stages), add new revision numbers to M and perform SVD again

Table 4.8: Time and resource concerns for impact analysis algorithms

	Pre-Processing	**Impact of Change Set**	**Post-Processing**
`PathImpact`	Running entire test suite	$O(f*T)$	Running entire test suite
`CoverageImpact`	Running entire test suite	$O(f*L)$	Running entire test suite
SDAA Technique	SVD ($O(F \log F)$)	$O(F*C)$	Occasional re-run of SVD

We used our technique on the system that had the worst performance for `PathImpact` to determine whether our technique could handle the systems that

PathImpact could not. The system that exemplified the worst case scenario for PathImpact was gcc, which took over 15GB of storage space and could not be completed due to excessive resource requests. The total space required while using our technique was 60MB, and the impact of a new revision could be calculated in approximately two seconds on a 2.2 Ghz computer. The initial time to seed the *M* matrix with the revisions was large (over four hours) due having to process 127,051 total revisions to the system. However, ideally the *M* matrix would be updated periodically during the course of development, which would minimize this time, taking no longer than 30 seconds for a given revision.

CoverageImpact scales better to larger systems than PathImpact does, mainly because the coverage vector is significantly smaller and less computationally intensive to maintain than the call trace required for PathImpact. CoverageImpact's complexity is not based on the depth of the call structure, but rather on the number of functions overall in the system, and thus does not increase in complexity as quickly as PathImpact. CoverageImpact, however, also could not be run effectively on gcc due to problems with creating static slices of the system [9].

4.6.2 PRIORITIZATION FROM EMPIRICAL DATA

While both CoverageImpact and PathImpact can be considered to be safe if their test suites exercise the entire code base, they both still produce false positive rates measured around 30% [36]. Our technique, however, is based upon historical information as to how files have been modified together in the past. Thus, our technique cannot be considered safe because files that are new to the system do not have sufficient historical data to be associated with any other file reliably. The advantage, however, in using historical data is that files that the most likely true positives appear in clusters with larger singular values and serves as a prioritization as to which files to check first for being affected by a new revision. When examining the true positives and false positives from `CoverageImpact`, `PathImpact`, and our SDAA technique from our industrial and open source case studies and the open source cases studies performed by Breech and Orso, we found that our SDAA technique was more likely to have a true positive (nearly 96% of the time) at the top of the list of impacted files. The other two techniques, with a comprehensive test suite, overall performed better at finding all the true positives. Figure 4.4 below shows the average true positive rate increases over the addition of files identified as impacted over the list of impacted files. As shown in Figure 4.4, our technique did find all the true positives in an impact analysis. A side effect of using historical data as opposed to semantic data is that our technique can only predict events that have happened before. Over a large set of revisions with a mature code base, our technique's performance improves significantly. If the system under inspection has had few revisions checked in to the source control system, the patterns of co-change that are detected through the use of SVD have not emerged yet. Thus, we would recommend our technique be used on systems that have been in development with a source control system for at least a few months.

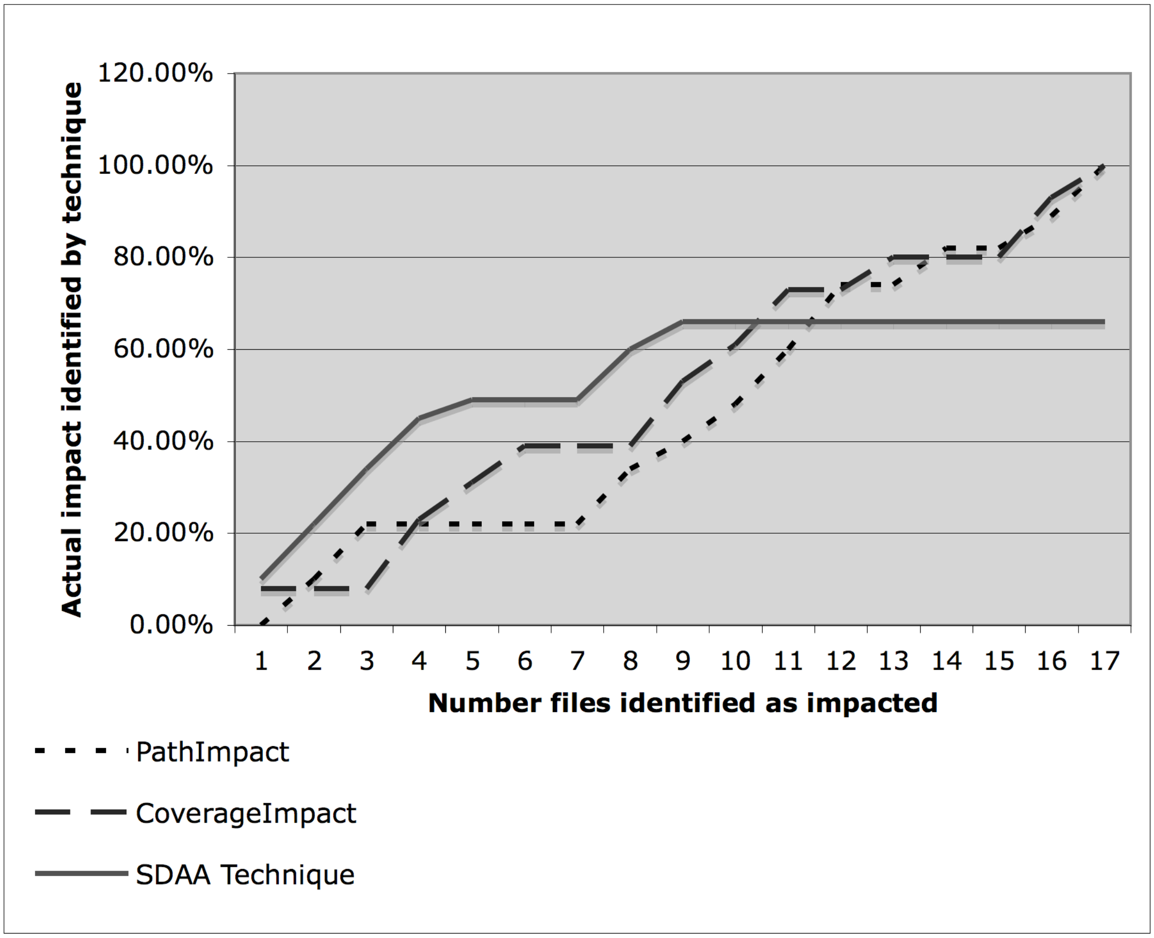


Figure 4.4: Comparison of ordering of true and false positives

4.6.3 NON-SOURCE FILES

In our industrial case study, we examined a system that had a large number of non-source files. We note that the impact of changing these files cannot be determined through the use of `PathImpact` or `CoverageImpact` due to their use of dynamic means or require manual intervention to examine the file, thus making the techniques more costly. Further, in some of the open source studies performed by Breech and Orso, these files are also present. For example, change log files, which are maintained by open source developers to let system users know what has changed in a given revision, are also maintained in the source control system for a number of these programs. While not modifying a change log file may not be a critical failure in some instances, in others it could lead a system user to misinterpret the functionality of the system, and thus use it in an improper way leading to a perceived failure. `PathImpact` nor `CoverageImpact` can identify the needed changes to the files, while our technique can.

4.7 CONCLUSIONS

In this chapter, we described how SDAA can be used to address the goal of software impact analysis. Our technique makes use of the information in change records to discover and define relationships between files within the system. The novel aspect of our technique as compared to other impact analysis techniques is the use of change records to drive an impact analysis that requires no access to the source code itself and also can incorporate all files in a system, even non-source files. In some systems, faults in non-source files can be just as severe as those in the code base [20]. Further, our technique utilizes historical evidence as to areas of the system currently under change to highlight files that are most likely to change together.

To examine the efficacy of our technique, we performed a feasibility study and case study with three releases of a product from IBM. The generated association clusters from the analysis were identifiable and directly relatable to specific requirements for each release or for an identified internal system component. The association clusters specifically illuminated areas of the code base where cross-component dependencies existed and components that included files that would not normally be examined in an analysis that used execution-based files, such as help files and configuration files. With enough information about how files change together, our technique has a confirmed true positive rate of around 60%. The other files in the impact set that are not confirmed true positives are either unconfirmed true positives or false positives.

In our comparative study between `PathImpact`, `CoverageImpact`, and our technique, we found that the system resource requirements for our technique were significantly less than those of `PathImpact`. `CoverageImpact`'s algorithm is as efficient as ours, however it and `PathImpact` both require instrumented execution information from either a set of test cases or from users of the system. Gathering this execution information could be time and resource consuming depending on the complexity of the test suite and/or system, and even then the test suite must be comprehensive to be of effective use. We also found that our technique was more likely than other techniques to recommend a true positive first in the list of impacted files due to our use of empirical data for prioritization.

CHAPTER 5

REGRESSION TEST PRIORITIZATION USING SDAA IMPACT ANALISYS

In the previous chapter, we described how SDAA can be leveraged with historical change records for the purpose of impact analysis. A natural extension of that research is then to apply that impact analysis to prioritize regression tests to determine if a fault has, in fact, been injected into the system. *Regression testing* is the process of retesting a system or component to verify that changes made to the system code have not caused unintended effects and that the system is still compliant with the specified requirements [2]. The goal of regression test selection (RTS) techniques is to isolate the tests most likely to uncover injected faults after a system modification.

We described in the previous chapter how we can leverage the similarity between an association cluster and a new set of changes to determine how likely a file has been affected by a change. We can extend this analysis by then determining which regression test best matches the affected association clusters.

Note that our goal is to create a regression test prioritization (RTP) technique as opposed to a RTS technique. The main difference between the two is that a RTS technique will return tests that are applicable in any way to the affected code in a new revision and could still contain some false positives. A RTP technique orders tests based on a set criteria, such as how many of the affected files it exercises or the severity of the faults, identifying the most comprehensive test first with the rest following. Our technique also identifies tests that exercise code that may be further away in the call chain from the affected source code, but the associated test is ranked further down the prioritization list. A high prioritization value on a test using our technique indicates that: (1) the files exercised by the test closely resemble the set of changed files; and (2) the set of changed files have a relatively high change frequency, indicating fault-proneness.

In this chapter, our goal is *to provide a methodology based upon our SDAA impact analysis technique that can prioritize regression tests to reduce the number of regression faults released to the field.* We will describe how one use of the SDAA can be leveraged and expanded upon to address additional development decisions through regression test prioritization. To examine the efficacy of our technique, we conducted an industrial case study with a project at IBM. Our RTP case study was conducted on the same data set as our impact analysis case study. We further evaluated our technique by comparing it to other RTP techniques.

5.1 BACKGROUND

The simplest RTS technique is the *retest-all* strategy, wherein the entire test suite is executed. However, techniques have been developed that can minimize the set of tests that need to be rerun while still maintaining the overall effectiveness of a retest-all strategy [45].

A RTS strategy is considered *safe* if the subset of tests identified by that strategy contain all the test cases from the test suite capable of finding faults in the system based upon a system modification [58].

One safe RTS strategy was developed by Rothermel and Harrold [45], called `SelectTests`. Their RTS strategy analyzes changes in the control dependence graph of the software system and changes to the test suite to determine the best subset of test cases. Rothermel et al. have shown that this technique is considered safe, identifying tests that cover not only changed lines but also newly-added lines of code. Further studies from Rothermel have shown that regression test selection and test case prioritization in general can be effective at reducing the amount of work needed to be done in testing after a system modification, but that the cost-effectiveness of these techniques can vary [47, 50]. Rothermel's technique, based upon his `PathImpact` impact analysis technique described in the previous chapter, can be potentially resource intensive if the program is complex.

One aspect of the technique developed by Rothermel et al. is that it requires source code to be effective. However, faults can be injected into non-source files in a system that might not be detected through a technique that solely examines source code. Consider a system that utilizes static properties files that are read in by different modules within the system. If a fault is injected into these properties files, there would be no reflection of this change in the source code, yet the fault could be as severe as any fault in the source itself [20].

Our technique adds to the body of knowledge of RTP strategies by addressing the problem of test selection when non-source files are involved. For example, 35% of all faults involved in one system examined in our study contained a non-source file. Further, we are trying to improve the overall cost-effectiveness of RTS strategies by using historical trends over time as opposed to processing dynamic call information.

5.2 SVD-BASED REGRESSION TEST PRIORITIZATION

In this section, we will describe the steps followed in the SDAA to generate a SVD-base regression test prioritization technique using the impact analysis technique portrayed in the previous chapter.

5.2.1 SDAA STEP 1: DETERMINE ANALYSIS GOALS

The first step in the SDAA is to determine the goal we are trying to reach through the use of development artifacts. In the previous chapter, we examined how we could leverage change records with SVD to determine the possible impact of a change. Our goal in this analysis is to *leverage the results of our SDAA impact analysis to further prioritize regression tests*.

5.2.2 SDAA STEP 2: PICK DEVELOPMENT ARTIFACT

As this RTP technique is based upon the impact analysis technique proposed in the previous chapter, the development artifacts are the same as those for our impact analysis technique, including the results from that technique when a new revision is made to the system. We begin by applying the impact analysis technique as previously described to determine the impact of a system modification. Thus, the actual development artifact is a list of potentially impacted association clusters. We then are interested in how these association clusters are linked to test cases in the system so that we can find the traceability between them. Test case traceabilities to files, or fault records indicating what files are executed during a test run, are required.

5.2.3 SDAA STEP 3: PICK GRANULARITY LEVEL

Again, following from the previous chapter, we begin at the file level, since that is the output of the previous technique. However, the output of this technique will be a prioritized list of tests to rerun. Thus, we will also be examining our results at the test case level.

5.2.4 SDAA STEP 4: IDENTIFYING DATA SOURCES

Source control systems are the primary source for gathering change records. When a developer checks a file in to a source control system, the system typically records the time of the check-in along with information about the developer and the nature of the change. Individual changes can be often be linked together into tracks, either through a specific mechanism in the source control system that records that information or through the examination of change record check-in information. With information regarding tracks, we are able to ascertain how files change together.

5.2.5 SDAA STEP 5: GATHER DATA AND BUILD M MATRIX

The next step is to use the current tracks in the system to determine how each test case in the system is related to a given association cluster from the impact analysis results. Using data from the source control and fault management system, we can map a track x directly to a particular fault in a one-to-one relationship. To accomplish this analysis, we can take the set of all tracks in the source control system and create a matrix T, with each row indicating a track and each column representing a particular file. An example matrix T has been constructed in Equation 5.1.

$$
T = \begin{array}{c} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \\ T6 \end{array}
\begin{array}{c} \begin{array}{ccccc} F1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \qquad \textbf{(5.1)}
$$

If each test case created one fault, the matrix T would also represent test case traceability to the files affected by those tests. However, in all likelihood, a test case can find more than one fault in a system. In this instance, tracks in T that are opened by the same test case can be combined. Thus, the matrix T represents which files are affected by each particular test case.

Multiplying matrix T by matrix R yields another set of associations in matrix P in Equation 5.2:

$T * R = [Tests\ x\ Files] * [Files\ x\ Clusters] = [Tests\ x\ Clusters] = P$ **(5.2)**

The P matrix that is generated from multiplying T and R together links a particular test case with its connection to a given association cluster in the same way that multiplying a single change vector x multiplied by R provided information on the impact of that track.

We can then relate the link between test cases and clusters back to individual files by multiplying P^T by another R, as shown in Equation 5.3:

$R * P^T = [Files\ x\ Clusters] * [Clusters\ x\ Tests] = [Files\ x\ Tests] = Y$ **(5.3)**

The matrix Y that is produced now has associated test cases in the system to particular files after transforming their relationship through the association clusters. This transformation is analogous to changing the basis of the original analysis matrix *M* by multiplying it by the results from the SVD.

This file association matrix *Y* is then used in conjunction with the fault information contained in the change vectors to provide an association between files and faults. If we multiply the matrix *Y* by a new change vector *x'*, the output is a single vector where each column is a particular test case and the values represent how closely the new change vector matches with each test case. An example of this multiplication is shown in Equation 5.4:

$$Yx' = \begin{array}{c} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \\ T6 \end{array} \begin{array}{c} \begin{array}{ccccc} F1 & F2 & F3 & F4 & F5 \end{array} \\ \begin{bmatrix} .69 & .09 & .23 & 0 & 0 \\ .32 & .90 & .76 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ .78 & .60 & .62 & 0 & 0 \\ .32 & .90 & .76 & 0 & 0 \end{bmatrix} \end{array} * \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} T1 = .32 & T2 = 1.7 & T3 = 0 & T4 = 0 & T5 = 1.2 & T6 = 1.7 \end{bmatrix} \quad \textbf{(5.4)}$$

The value in the result vector itself represent how close the new modification matches a given test case, taking into account files that tend to change together. For instance, a high-priority test case will not only contain files that were directly changed in the new modification, but will also contain files that tend to be affected when the newly-modified files change. The values themselves are used as a relative prioritization among the test suite for this modification and this modification alone as opposed to using the prioritization values as an absolute that can be compared against different modifications. The possible maximum value for a test in the prioritization vector is proportional to the number of files that have been changed in a revision. However, prioritization vectors can be additive if several system modifications are being examined simultaneously. In this instance, we find that T2 and T6 are equally appropriate to run in this instance, while T3 and T4 are not related at all to the changed files.

Test cases that are associated with non-zero values in the result vector execute areas of code that have previous problems and are candidates for being rerun. These test cases are ones that either directly involve the files that have been affected by the change or files that are in the same association clusters as the affected files. Thus, files that are closely tied to the affected files through previous system modifications, will have a greater impact on the values in the result vector.

5.2.6 LIMITATIONS

Our RTP technique has the same limitations as the impact analysis technique on which this RTP technique is based, such as the safeness of the technique and reliance on non-opportunistic changes. Another limitation is that our technique requires that traceability exists between test cases and the faults that are discovered by the test cases. This traceability is required so that a matrix of files affected by test cases can be compiled.

5.3 INDUSTRIAL CASE STUDY

Research in our RTP technique was performed at IBM in Durham, NC. In this section, we will discuss our industrial case study that evolved from our investigations into impact analysis. Our hypothesis is that *a methodology based upon SVD using historical change records can be used to prioritize regression tests.*

5.3.1 CASE STUDY SETUP

We selected Matlab 7.2 R2006a as our SVD tool and began by examining available data sources. IBM's source control and fault management system generates detailed logs on tracks and both pre- and post-release failures. We continued our study on the same data set as our impact analysis case study.

5.3.2 SVD-BASED REGRESSION TEST PRIORITIZATION

The prioritization approach through our technique identifies related files and includes them in the regression test prioritization based upon the weighting values given to files in clusters by the SVD in the Y matrix generated from the U matrix. The value a file has with a given cluster in the Y matrix dictates its impact within a regression test prioritization as well. What this yields is a prioritization list in which the test cases are ordered as to not only how similar the track is to a given test case, but also the impact that a test could have on surrounding files in an association cluster, as described in Section 3.6. A high prioritization value on a test indicates that: (1) the files exercised by the test closely resemble the set of changed files; and (2) the set of changed files have a relatively high change frequency, indicating fault-proneness.

We evaluated the efficacy of this methodology by using a random data splitting technique with the three minor releases of an IBM software system. We began by randomly selecting two-thirds of the tracks for each release as the "historical data" from which we generated the association clusters. From the remaining one-third of the tracks, twenty tracks were selected for use as "future changes" that are being introduced into the system and for which we are prioritizing regression tests. The remaining tracks in the one-third set represented the future set of faults that would be found in the system. For example, if simple retest suggests we retest File 1 and our technique further suggests we test Files 1 and 2, if there are instances of Files 1 and 2 together in the "future set," we assume our technique retest would have prevented that fault.

For each of the twenty tracks selected to represent new changes introduced into the system, we converted these tracks into vectors. Each of these vectors was then multiplied by the association matrix Y generated from the "historical data" set of tracks. Since the association matrix Y links test cases to files, the results of this multiplication yielded a regression test prioritization list. We compared the prioritization lists from the twenty selected "future change" tracks with the future set of faults to determine if the OUR TECHNIQUE prioritization list identifies additional test cases that actually appears in the future set. If the suggested tests did appear in the one-third future set of data, then it was recorded as a positive match recommendation for the regression test prioritization model. However, an identified test could be a true positive and not be found in the one-third future set if it was not run by the team. Thus, the results we present here are minimum possible values and are, in all likelihood, somewhat higher in actuality. We repeated the data splitting analysis six times. The results of our analysis are found in Table 5.1. Figure 5.1 shows the percentage of the total number of tests that were recommended for each "future change" track.

Table 5.1: Model recommendations

Recommendations	Percentage Non-Source Files	Original test only	Confirmed True Positive
Original test only	35.2%	50%	N/A
15-50 additional tests	33.6%	19%	25%
1-14 additional tests	39.5%	31%	40%

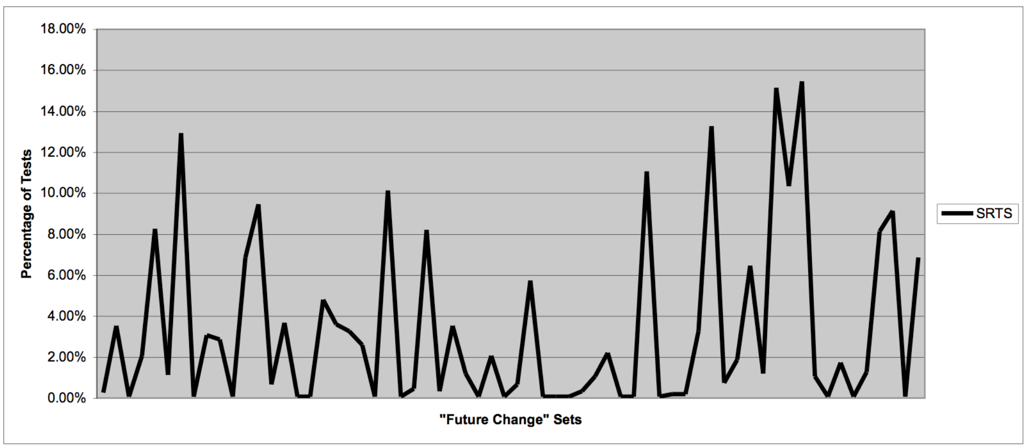


Figure 5.1: Percentage of tests recommended to be rerun

We found that 50% of the time our technique did not recommend any additional tests beyond a simple retest of changed files. These instances are represented in Figure 5.1 by the points that appear close to 0%. If the model recommends the same tests as a simple retest, this indicates that the areas of the system that these tracks represent are either historically self-contained modules or that there is not enough change data among the files to make an accurate prediction. In the case of the self-contained modules, a regression set that is the same as a simple retest is a positive result, since testing can be expensive, and we do not want to incur additional cost with little extra benefit.

Conversely, if there is sparse change data in a section of the code, where numerous files have only changed once or twice together, a cohesive association cluster structure might not have formed yet. In these "loose" cluster areas of the system, our model recommends more tests with an overall lower confirmed true positive rate because of the number of recommended tests. These instances are represented by the spikes in the graph in Figure 5.1. However, we consistently found that the tests at the top of the prioritization list were confirmed true positives. Approximately 60% of the first test cases identified a future fault in the files included in a new set of changes.

The percentage of the time that a prioritized test finds a fault is found in Figure 5.2. The tests that appear at the top of the prioritization list also did not appear in the simple retest about one-third of the time, due to the initial test being a test that exercised code that was directly related to the affected code, but not actually a part of the affected code.

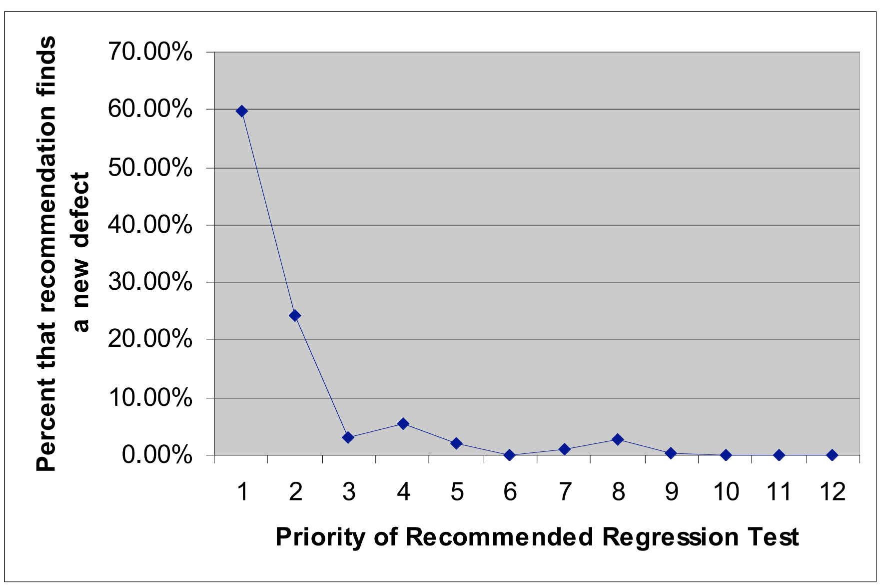


Figure 5.2: Percentage matches in future data by regression test priority

5.4 COMPARISON TO OTHER TECHNIQUES

In this section, we will perform a theoretical comparison of our technique with other RTS techniques. Rothermel and Harrold created a framework for evaluating and comparing RTS techniques and have used their framework on a representative set of techniques [43]. We use their framework to show how our technique adds to the body of knowledge in regression testing. Rothermel and Harrold use four characteristics to describe and evaluate a RTS technique:

- **inclusiveness:** whether a RTS technique is considered safe or not, thus finding all true positives in a regression test selection[9];
- **precision:** while inclusiveness measures how effective a technique is at finding all the correct tests for a regression test run, precision measures how many extra, unneeded tests are recommended by the technique that do not add any benefit;
- **efficiency:** the computational and overhead costs associated with running the RTS technique; and
- **generality:** the context for which the RTS technique can be used, including language, and program constructs.

5.4.1 INCLUSIVENESS

The inclusiveness of our technique is based upon the inclusiveness of our impact analysis technique. Our regression test prioritization is not considered safe because of our use of historical data in our impact analysis aspect in the technique. If a new system modification is introduced in which sets of files change together that never have changed

[9] Rothermel and Harrold do note that a RTS technique can only be as effective as the test suite in revealing modifications and faults.

together before, such as a feature being introduced in files that were previously not in the system, then our technique might omit some tests. The limitation of not being safe is one that cannot be avoided for historically-based RTS techniques, due to the nature of the data that is being used for the technique [43].

According to Rothermel and Harrold, there are only four truly safe RTS techniques available: their technique based on path analysis; Laski and Szermer’s method based on encapsulating code areas by call analysis; Leung and White’s firewall-based technique; and Fischer’s linear equation technique [43, 46]. These four safe techniques are all similar in that they select all tests that traverse any part of the call path of an affected area of code. All of these techniques use a method building call and flow graphs to determine every execution path that could possibly be affected by a change, and then select the tests that intersect those paths. However, changes can take place outside of the code base that can have an effect on the system as a whole. For example, a change such as the one shown in Figure 5.3 could not be detected by any of these four safe techniques based on call analysis, but could be detected by our SDAA technique.

`Image.displayImage(“openMail.jpg”);`	`Image.displayImage(“openMail.jpg”);`
	

Figure 5.3a: Version 1 **Figure 5.3b: Version 2**

Figure 5.3: Change in non-source file example

When the file *openmail.jpg* is changed and submitted as a revision to the source control system, our impact analysis technique would be able to determine that there has been a change to this media file. While this change certainly affects the file in which the above code would appear, it would also indicate other places in the system that possibly use that file. Rothermel and Harrold's safe technique based upon `PathImpact` is considered safe with respect to the source code and will identify any test case that exercises affected code. However, any change that falls outside of source code and has a manual test case could not be detected by Rothermel's technique.

5.4.2 PRECISION

The precision of our technique improves as the number of changes in the system increases. Rothermel and Harrold recognize that no RTS technique can be guaranteed as 100% precise [43]. Fewer tests are identified for regression test and the number of matches that those tests have in the random one-third data set increase. A Pearson correlation between the number of changes of the files involved in a new track and the accuracy of the model ($p = .003$) indicates that as more change data is collected on a given set of files, the better the RTP for those files. Thus, our technique's precision is only governed by how well future modifications match historical patterns. If new features are not being added to a system and new modifications follow historical patterns, then the precision of our technique could be high.

The four safe regression techniques mentioned above in particular can have the problem of being imprecise. When selecting every test that intersects a given call path, numerous extra tests can be selected in the pursuit of ensuring that the result set is safe.

Laski and Szermer's RTS technique identifies a control scope of each decision statement and finds every call path in which that statement exists. They refer to these closures of call paths as clusters, and a given cluster can have numerous tests associated with it. However, if only a small portion of a cluster is changed, all the tests for that cluster are still selected. Rothermel and Harrold's technique based on `PathImpact` works in a similar fashion. These techniques have shown result sets where the number of false positives approaches 40% of the set [47].

There is a tradeoff between inclusiveness and precision, as the fully inclusive (safe) techniques also tend to have higher numbers of false positives, while those techniques that try to eliminate as many false positives as possible also exclude some true positives at the same time [43]. We recognize that our RTP technique can never be safe due to the use of historical data, but through the use of the historical data, we can improve the precision of our technique. The prioritization vector produced by our technique orders the test cases from those that are most likely to be true positives based on historical evidence. Thus, the precision of our technique is governed by how far down the prioritization list a developer or tester goes, with a degree of diminishing returns as the list progresses. Further, since our technique is based on historical evidence and not actual semantic data, the prioritization may be better categorized as a "recommendation list," as opposed to a definitive list of required regression tests. As shown above in Figure 5.2, the likelihood of a recommendation being a true positive is higher at the top of the list than further down. We found that in the top seven recommendations to have nearly 95% precision. If developers are looking for the most relevant tests quickly, then our technique would be a viable possibility.

5.4.3 EFFICIENCY

Section 4.6 dealt largely with the efficiency of the impact analysis algorithm that is used in our RTP technique. The key difference between our technique and most other RTS techniques is that our technique does not incorporate any semantic or dynamic execution information. Our technique requires the gathering of development artifacts from a data source, performing a SVD, and then interpreting the results. As discussed in Chapter 4, the most inefficient part of our technique is the computation of the SVD, assuming that there are automated procedures in place to gather change records to populate the *M* matrix. However, the SVD does not need to be calculated with each new system modification, and that process could be easily automated to run during off-hours. We believe that it would be sufficient to compute a new SVD at the end of the day automatically after no more modifications would be made. In this way, the efficiency of our technique can be much higher than any dynamic RTS technique, since this would reduce our technique to a simple matrix times a linear vector for each RTS recommendation. In an organization where dynamic means are infeasible, such as instances where call graphs cannot be created and maintained effectively, our technique might be more appropriate.

However, our technique suffers from a more software development process-intensive requirement than other RTS techniques. Traceability of tests to the files they execute is required to generate the prioritization list from impact analysis results. In our industrial case study, traceability information was recorded as a by-product of the testing process and was thus readily available with no additional overhead. Other organizations, however, may have to adjust their process to gather this information, or run a code coverage tool to gather the data. Other static RTS techniques suffer from this limitation as well. Srikanth's PORT methodology required the careful maintenance of customer and developer information regarding requirements as well as which pieces of code were linked to particular requirements [57].

We must note that certain development traceabilities must be maintained for the overall efficiency of our technique to be better than theirs. In most RTS techniques, the pre-processing required for the technique, whether that be executing instrumented code, gathering a call trace, or performing static code slicing, is the most time intensive. The algorithms themselves to then determine the regression tests are relatively comparable operating in linear time.

5.4.4 GENERALITY

The novel aspect of our RTS technique is its generality in that its context is all-inclusive. Any file that is managed through a change management system, whether that file is source code, media files, documentation, or anything else, our technique can recommend tests appropriate for those files based upon which other files they have changed with. Faults that are found in non-source files can be as severe as those within source files and thus we believe that our technique for prioritizing tests with regard to all files in the system can provide some added insight into prioritizing regression test cases.

Our interpretation of generality is somewhat broader than Rothermel and Harrold. They portray generality for RTS techniques being the applicable to different programming languages, environments, and testing methods. RTS techniques based on non-semantic or dynamic information effectively change this definition by changing the overall context in which the RTS technique is applied. Instead of operating on the source code and test code itself to determine the regression tests to rerun, techniques such as our SDAA technique and Srikanth's PORT technique operate related, but completely separate, means. By changing the context of the RTS methodology to non-source means, there are no restrictions on programming language, environment, or testing methods. However, it does add a separate level of constructs required for execution, namely various types of traceabilities between the code and the development artifacts of change records for our technique or requirements for PORT.

5.5 CONCLUSIONS

In this chapter, we explored an empirically-based regression test prioritization method based upon structures discovered through change records and singular value decomposition. Our technique makes use of the information in change records to discover and define relationships between files within the system. These relationships, called association clusters, can then be utilized within the development process identify areas of risk in the system that are currently under change.

To show the efficacy of our technique, a case study was performed with three releases of a product from IBM. The generated association clusters from the analysis were identifiable and directly relatable to specific requirements for each release or for an identified internal system component. The association clusters specifically illuminated areas of the code base where cross-file dependencies existed and areas of the system that included files that would not normally be examined in an analysis that used execution-based files, such as help files and configuration files. We performed a post hoc case study using this technique with three minor releases of a software product. We found that our methodology suggested additional regression tests in 50% of test runs and that the highest-priority suggested test found an additional fault 60% of the time.

Our technique adds to the body of knowledge in RTS by providing a method that extends the generality of RTS outside of the realm of programming languages and environments by using development artifacts, somewhat similarly to Srikanth's PORT methodology. Our technique, while not safe like many other RTS techniques, improves its precision by prioritizing regression tests based upon historical evidence regarding previous fault-proneness. Overall, our technique can be much more efficient than other RTS techniques given that traceability information is readily available through the development process. If traceability information is not readily available, it can be generated through code coverage tools, which can add extra overhead to our technique.

CHAPTER 6

STATIC ANALYSIS OF DEVELOPMENT ARTIFACTS

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [19] without execution of the code. Static analysis tools search for implementation problems identified by a predefined set of rules of potential anomalies commonly found in source code. The static analysis rules range from possible mistypes in the code (e.g. = instead of ==) to more complex errors in the system logic (e.g. memory leaks). We term the use of static analysis tools as *automated static analysis (ASA)*. An *ASA alert* is a single report from ASA, indicating an area of the code base that has broken an ASA rule. Each ASA alert has an *ASA alert type*, which describes the ASA rule that has been broken.

One of the more difficult aspects of dealing with and utilizing static analysis results comes from the amount of data that ASAs produce [64]. For example, the number of ASA alerts in a software component tend to be correlated with the number of files in that component [64]. In an experiment with ASAs Splint[10] and PolySpace[11], the tools generated a buffer overflow alert for every 14-46 lines of code for three open source applications [64].

[10] http://www.splint.org/
[11] http://www.polyspace.com/

Thus, for an ASA using multiple rule types on a large software system, the number of generated alerts can become too large for a development team to reasonably inspect. If the number of reported ASA alerts gets too large, the task of examining every ASA alert can overwhelm the team, which takes time away from other development activities and can ultimately cost more in resources than not using ASA at all.

The main reason for this increase in cost is because not every failure in the system will be related to something discovered by an ASA alert type. Research has shown that there can be as many as 50 false positive ASA alerts for each true positive [40]. A *false positive* is an alert indicating a problem in the source code when there actually is no fault. False positives in ASA tools can be generated due to unspecific ASA rules. Rules that are easily matched throughout the code base can often generate excessive alerts. Further, even a true positive alert may never lead to a failure in the system. If the ASA alert is in a section of the code that is rarely or never executed, it may not be a good business decision to expend extra resources to repair the code associated with the alert.

Despite these potential cost and effort concerns with ASA tools, research has shown that ASA alerts can identify certain classifications of faults and field failures [62]. The challenge in using these tools is to find a cost-effective method for eliminating false positive alerts and alerts in sections of the code that do not need to be fixed. A number of commercial ASA tools have built-in heuristics that allow the tool to "learn" where to suppress certain kinds and locations of ASA alerts. Research is also being performed in filtering ASA alerts [16] to limit the viewing of alerts to just those of immediate importance to the developer.

Using SDAA, our goal is *to provide a methodology for distilling the results from ASA tools to provide more directed guidance for developers and managers in leveraging the information from ASA*. In different software projects, various specific groups of ASA alerts are going to be more likely to appear with a reported system failure than others. Using SVD, files that contain groups of static analysis alerts historically associated with field failures can be identified. Relationships between historical field failures and change records in conjunction with ASA alerts can be used to generate *ASA alert signatures*. An ASA alert signature is a set of static analysis alert types that has historically been associated to one or more field failures in a particular project. The SVD in SDAA provides a means for associating files with field failures and ASA alerts with those files. A set of files that has changed together is identified as failure-prone if those files contain all of the alert types in an ASA alert signature.

In this chapter, we discuss how SDAA can address goals related to directing development effort when working with static analysis results and our efforts in evaluating our technique. We first performed a feasibility study on a small subsystem from the large industrial project from IBM discussed in previous chapters to evolve our technique. We expanded our investigation to six major components of the same software system across six builds over a period of eighteen months. The results from this study were compared with Nagappan and Ball's ASA filtering technique for identifying failure-prone files and modules.

6.1 BACKGROUND

In this section, we will discuss background information on ASA and other research in filtering ASA results to guide development decisions.

6.1.1 AUTOMATED STATIC ANALYSIS

Rutar et al. [51] compared various automated static analysis tools and the techniques they employed to detect faults. These tools included FindBugs [17], JLint[12], PMD[13] , Bandera [12], and ESC/Java [11]. Each of these tools are used to analyze static source code or bytecode (sometimes with additional notations in the case of Bandera and ESC/Java) to locate common programmer mistakes. Rutar also investigated the general classifications of the kinds of mistakes that these tools could locate. Each tool found a different subset of common programmer mistakes, including such problems as null pointer dereferencing and division by zero. More complex problems could also be found, such as unclosed data streams. Each tool had its strengths and weaknesses. A common weakness among all of the tools, however, was the number of alerts reported and the number among those that are false positives. Rutar comments that some mechanism for filtering out false positives is necessary to make the tools more useable and their results more accessible [51].

6.1.2 USING ASA TO SEPARATE HIGH- AND LOW-QUALITY COMPONENTS

ASA alerts have been used to identify fault-prone areas of code. Nagappan and Ball performed a study in which static analysis alerts were used to predict the pre-release fault density of Windows Server 2003 [32]. Nagappan used two internal Microsoft static analysis tools, PREfix and PREfast. While developed separately, the two tools use similar

[12] http://artho.com/jlint
[13] http://pmd.sourceforge.net

methodologies to identify problematic sections of code, with the main difference being that PREfast is less thorough and is intended as a quick checking tool. Nagappan gathered data on the ASA alert density and pre-release fault density for 199 components of Windows Server 2003. The research demonstrated a positive correlation between the ASA fault density and pre-release fault density and that discriminant analysis of ASA faults could be used to separate high- from low-quality components with 83% accuracy. Further, Nagappan and Ball used the PREfix and PREfast data from two-thirds of the components to build a linear regression model to predict the fault density of the remaining one-third of the components. The results of their study showed that PREfix and PREfast ASA alerts could be used in-process to separate high quality components.

Another study was conducted by Zheng et al. on the use of static analysis at Nortel Networks, Inc. [34, 62]. ASA and failure data from three products (over three million lines of code) that underwent ASA during test were analyzed [34, 62]. The data demonstrated a statistically-significant correlation between the number of ASA alerts and field failures in a module (a grouping of files). These results indicate that when a module has a large quantity of ASA faults, the module is likely to be problematic in the field, information that can be used to prioritize validation and verification (V&V) efforts prior to release. Finally, discriminant analysis indicated that ASA faults could be used to separate fault-prone from non-fault prone modules with 87.5% accuracy.

In our research, we compare our methodology for identifying high-quality components to the work of Nagappan and Ball and Zheng et al. One of the main differences in our technique is that these previous techniques only utilized the quantity of ASA alerts. In our study, we use information about the types of the alerts that appear in groups of files that have historically changed together to identify failure-prone files. Further, these two previous studies were performed at the module level. Due to the availability of higher granularity data, our study is performed at the file level.

6.2 GENERATING ALERT SIGNATURES

In this section, we will describe the steps in the SDAA for identifying failure-prone files using static analysis alerts. The process begins in a similar fashion to the impact analysis process, except that we will only use field failures to build the association clusters. Our static analysis technique, however, adds a second layer of development artifacts (alert types), which are integrated into the technique through a second *M* matrix. We will also describe how to apply the generated alert signatures to a set of ASA alerts types and system files of a future build.

6.2.1 SDAA STEP 1: DETERMINE ANALYSIS GOALS

The first step in the SDAA is to determine the goal we are trying to reach though the use of development artifacts. Static analysis tools are designed to provide detailed information about potentially problematic sections of code during development. The ASA alert information can help direct the decisions of managers and developers as to which areas of the code base require the most attention to maximize the benefits of the time spent repairing the code. However, one of the main difficulties of using ASA is that the amount of information from the tools can overwhelm developers, thus making the decision of where to target their efforts more difficult. *The goal in this analysis is to separate out high- and low-*

quality files in the code to help developers make the decision as to where to direct their fault removal efforts.

6.2.2 SDAA STEP 2: PICK DEVELOPMENT ARTIFACT TYPE

We begin with the set of alerts that are provided by an ASA tool. However, we are most interested in those alerts that are tied to sections of code that have led to field failures in a prior release. Thus, we also require field failure artifacts and change record artifacts to identify the files that are linked to each field failure. These three sets of development artifacts will allow us to follow the traceability from groups of static analysis alerts that tend to be linked to field failures.

6.2.3 SDAA STEP 3: PICK GRANULARITY LEVEL

The ASA tools usually generate alert information at the file level and often down to the line of code. Similarly, change record artifacts are often available at both the file level and line of code level. Field failure information is more often going to be at the file level. While it is possible to combine the data sets together to perform this analysis at the module or component level, field failures can often span multiple modules in the system. Expanding this particular analysis to a lower granularity level may obscure the results of the signatures, providing general signatures that match to more areas of the system than appropriate, thus creating extra false positives.

6.2.4 SDAA STEP 4: IDENTIFY DATA SOURCES

Historical records of field failures, change records, and static analysis results are all required to generate accurate ASA alert signatures. Further, there needs to be traceability links between field failures and the change records created during the correction of those failures. Because of this reliance on accurate historical records, our technique is best used in a development environment where process information is recorded in a systematic way and is available for use by the development teams. In our previous experience in gathering change data as described in previous chapters, we have found that frequent code check-ins to source control systems of relatively small size (three to five files) improves the performance of our technique significantly as the SVD is better able to isolate co-changing areas of code. Thus, if a systematic methodology is not being followed by a development team, the practice of continuous integration [5] is sufficient.

6.2.5 SDAA STEPS 5-7: GATHER DATA AND BUILD M MATRIX

As mentioned above, our static analysis technique uses the same initial steps as our impact analysis technique, except that instead of using all changes to build the *M* matrix, we only use those changes that are associated with field failures. Using only field failure changes, we can determine where field failures tend to appear in the code base. These steps can be referenced in Section 4.2.

6.2.8 SDAA STEP 5: GATHER DATA AND BUILD M MATRIX WITH ASA ALERTS

We return to Step 5 in the SDAA to identify the traceability link between groups of ASA alerts and files involved in field failures. Once we know what files are strongly associated with field failures, we can then determine how the ASA alerts compare with these file clusters. In this step, we will create a new matrix *M*. However, this matrix will be an asymmetric matrix with the previously generated clusters on one axis and the different types of static analysis alerts on the other. The values in the matrix *M* will be the difference in the number of ASA alerts found between two builds. We are interested in the difference between two baselines because this will highlight any possible correlation between the removal of ASA alerts with fewer field failures and visa-versa. An example matrix M is shown in Equation 6.4.

$$M = \begin{array}{c} \\ A1 \\ A2 \\ A3 \end{array} \!\!\! \begin{array}{c} \begin{array}{ccccc} C1 & C2 & C3 & C4 & C5 \end{array} \\ \begin{bmatrix} 95 & -40 & 3 & 0 & 25 \\ 25 & 60 & 26 & 17 & 0 \\ 0 & 0 & 34 & 89 & -45 \end{bmatrix} \end{array} \qquad \textbf{(6.4)}$$

6.2.9 SDAA STEPS 6 AND 7: PERFORM SVD ON M MATRIX WITH ASA ALERTS

Performing a SVD on the new matrix M yields another set of U, S, and V matrices. We can interpret these matrices in much the same way as before. However, now the columns of U indicate clusters of ASA alerts and the rows of U indicate alert types. The singular values in the S matrix are interpreted slightly differently than with clusters involving files and field failures. In clusters involving ASA alerts, a high singular value indicates that that set of alerts appears numerous times within the code base.

In some instances, however, an alert signature will be comprised of a single alert type with a high singular value. This phenomenon is indicative of an alert type that is so pervasive in the code base that it can match with nearly every file, thus eliminating much of the benefit of our technique. These types of alerts can be identified if they appear in a signature by themselves and have a singular value that is a factor of ten higher than the next value. We eliminate these signatures from our technique, since this would significantly increase the number of potentially failure-prone files identified by our technique that may not contain actual faults.

An opposite problem can occur with ASA alert signatures with low singular values. For example, an ASA alert signature consisting of a subset of alerts from another ASA alert signature with a lower singular value can appear. In this instance, the signature with the lower singular value would match with fewer field failures, thus identifying only a few files as being failure-prone. Often, these files will have already been identified by another ASA alert signature with a higher singular value. Thus, the goal then is to choose a set of signatures that are distinct from each other, but also not too general or pervasive (as indicated by the singular value).

6.2.10 SDAA STEP 8: APPLYING THE CLUSTERS TO THE ANALYSIS GOAL

After the ASA alert signatures have been generated and identified, each subset of ASA alert types found in a given signature can be compared to a full set of ASA alerts from a code base. However, since these ASA alert signatures were generated based on clusters of files, the signatures need to be applied in a similar fashion. The ASA alert signatures are linked to groups of files that were consistently changed to repair field failures. If we compared an ASA alert signature to individual files, the signatures would return fewer matches, since a file may not have all of the alert types present within it.

Clusters of files in the system are generated in the same way as described in Steps 6 and 7, except that we examine all changes in the system up to the latest build date that were made to modify faults and field failures, as opposed to just those changes made to repair field failures. These clusters represent previous error paths in the system that have been corrected and how files tended to change together over time. Once these clusters have been generated, the alerts contained in each cluster at the later build date can be gathered based upon the files within each cluster. ASA alert signatures can then be compared with the alerts associated with each file cluster to determine which areas of the system may require further V&V efforts.

6.2.11 LIMITATIONS

The main limitation on selecting an appropriate data source is the analyst's confidence in the accuracy of the dataset. The analyst must be convinced that when a set of changes are attributed to repairing a given fault or failure, those changes are indeed associated with that fault or failure and are not associated with another set of changes. Another way of interpreting this is that there can be no opportunistic changes, such as if a developer corrected two faults at once and associated all the corrections with one fault. Minimizing opportunistic changes is essential for establishing confidence in the data source. If opportunistic changes occur during fault removal efforts, we cannot be certain that the set of files that change together are indeed related to a particular fault.

In an open source project, for instance, we could not guarantee that files that are checked in together are linked in any way, since a developer could address several faults before checking in a set of code. The analyst must determine whether there are acceptable data sets available in which changes for addressing faults are accurately recorded. We will address this limitation further later in the chapter.

Another limitation of this technique is that even at its optimal rate at identifying failure-prone files, our technique cannot improve upon true and false positive rates of the static analysis tool selected. Research has shown that ASA tools can only find certain types of programmer errors [62] and, thus, cannot be expected to find all faults that lead to field failures. Further, our technique is specifically based on historical data. Therefore, files that are recently created and have little to no change history or files that have not in the past been failure-prone cannot be isolated effectively through our technique. However, the goal of our technique is to guide developer actions, providing information as to where field failures are most likely to occur.

6.3 INDUSTRIAL FEASIBILITY STUDY

We performed a feasibility study of our technique with a large industrial software system to help evolve our technique. *Our hypothesis is that automated static analysis alert signatures generated from historical information through singular value decomposition can identify files that are the likely to contain field failure-causing true positive alerts from the ASA tool.*

6.3.1 FEASIBILITY STUDY SETUP

We selected Matlab 7.2 R2006a as our SVD tool and used an internal ASA tool for generating ASA alerts. The internal IBM ASA is called Bugs, Errors, and Mistakes (BEAM) [8]. BEAM uses data flow analysis to analyze C, C++, and Java code as part of the compilation process. One of the main goals of BEAM is to avoid as many false positives as possible while not requiring any extra specification from the user. To do this, BEAM spends extra execution cycles traversing paths that it identifies as leading to an error to ensure that the path is indeed executable. Because BEAM requires extra execution cycles, the tool is intended to be run overnight on the day's build [8].

BEAM reports 74 different ASA alerts and classifies the ASA alerts into five separate categories: error, mistake, warning, security, and portability. An ASA error alert is a high priority alert, with mistake and warning as medium and low priority, respectively. Security alerts indicate areas where the program may be subverted, such as unverified inputs. Portability alerts are for problems that would only appear if the code is ported to another machine with a different bit depth (such as 32-bit to 64-bit). Each classification of alert can be enabled or disabled according to the developer's preferences.

We performed our case study on three modules of a large IBM project. We selected these particular modules (totaling 5,244 files) because they were primarily written in C and C++, which are two of the languages that this particular version of BEAM could analyze. Two builds of the software project were chosen that were approximately one year apart, the first from late October 2005 and the second December 2006. Clusters of files based on field failures were created from field failures and change records for the first build between October 2005 and December 2006.

BEAM was run on each release of the software, and we gathered information on the files, alert types, and line numbers where the alerts appeared. All 74 alert types from the five categories of alerts were included. ASA alerts were associated with clusters of files and the difference between the two releases was calculated for generating the ASA alert signature clusters. Once we built our ASA alert signatures, we applied them to the December 2006 build to predict potential failure-prone files between then and March 2007. We are using field failure information from the three months after the December 2006 releases as our actual result.

6.3.2 ASA ALERT SIGNATURES

Using the field failures data gathered from the October 2005 and the December 2006 builds of the system along with the changes in ASA alerts over that time period, we generated a total set of 74 ASA alert signatures. The SVD generates the same number of clusters as the rank of the original matrix and thus this is why 74 alert signature clusters were created. A graph of the singular values of the 74 ASA alert signatures can be found in Figure 6.1.

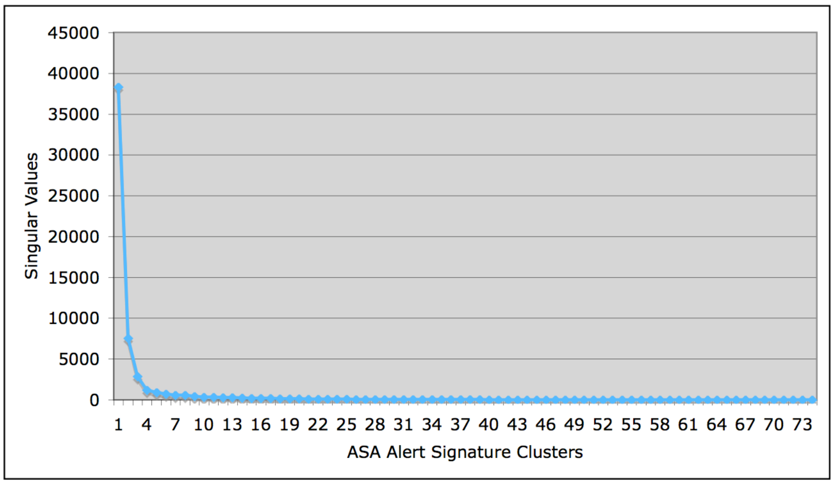


Figure 6.1: ASA alert signature cluster singular values

Note the steep curve in the graph among the first four ASA alert signatures as a visual indication as to how many signatures will be useful in our technique. In the following sections, we will discuss the first five ASA alert signatures, which explain over 93% of the overall variability in the cluster set. In other words, if we reconstructed the original M matrix using only the data corresponding to the first five ASA alert signatures, the result would be within 7% of the actual matrix M. The remaining 7% of the variability, found in the other 69 clusters, are all either supersets of the first five clusters or contain alerts that do not appear often or at all in the same files as field failures. The top five signatures are discussed below.

ASA Alert Signature 1: A Single Warning

The first ASA alert signature, associated with the largest singular value of 38,551, consists of over 69% of the total variability within the matrix of alert types. This ASA alert signature is comprised of a single, minor ASA alert that is commonly found throughout the code base. The singular value associated with this signature indicates a cause for concern when applying this particular signature to a set of ASA alerts. With such a high singular value and simple signature composition, this type of ASA alert signature could match up with nearly every part of the system, both failure-prone and free from failure. This particular ASA alert type consists of 35% of all alerts reported and is found in nearly every file that contained ASA alerts. Because of the pervasiveness of this alert type throughout the system, it does not provide much information regarding failure-prone files. We examined this alert type in correlation with field failures, as was done by Zheng et al. [62] and Nagappan and Ball [32], and found that for our three modules, there is not a strong correlation between this alert type and field failures. Due to the high singular value and pervasiveness of this alert type, we exclude this cluster from our set of ASA alert signatures.

ASA Alert Signature 2: A Misstep in the Path

The second ASA alert signature has a singular value of 7,551 and consists of 13% of the overall variability from the matrix of alert types. This set of ASA alerts focuses on missing parts of an execution path. The signature consists of five alert types. The alert types are as follows:

Mistake 5: *Expression always evaluates true or false*

Warning 5: *Operator "=" in the Boolean expression should possibly be "=="*

Warning 13: *Function never used*

Portability 2: *The cast (int)long will cause truncation on the target machine*

Security 2: *Passing untrusted input to argument*

The first two alert types, M5 and W5, are particularly linked together, where the latter is a specific instance of the former alert type. Another alert type, W13, warns that there is a function in the file that is never called within the scope. This type of problem is one that could arise from an *if/then* test that always registers true or false and thus both parts of the execution path are not used. The other two alert types deal with portability and security issues. P2 indicates a place where truncation could happen if compiled on a different platform, while S2 highlights an instance where an untrusted input is passed into a function.

ASA Alert Signature 3: Common Errors

The third ASA alert signature has a singular value of 2,875 and consists of 5% of the overall variability from the matrix of alert types. The alert types are as follows:

Warning 15: *then/else/loop not surrounded by braces*

Warning 16: *Function accesses the same variable through two parameters*

Mistake 18: *Comparing pointers to strings, not the strings themselves*

Error 18: *Function lacks a return statement with a value*

This set of ASA alerts links together four different common programming errors that can lead to field failures. These alerts can lead to the program taking unexpected paths through the code because of incorrect scope or a garbage value returned from a function. An incorrect comparison can also lead to the system interpreting an input differently than expected.

ASA Alert Signature 4: Memory Leaks

The fourth ASA alert signature has a singular value of 1,204 and consists of 2% of the overall variability from the matrix of alert types. This set of three ASA alert types focuses on memory leaks. The alert types are as follows:

Error 23h: *Heap memory leak*

Warning 9: *Return of function not used*

Mistake 21: *Advisory has been issued for this function*

The ASA alert type for memory leaks, E23h, is the most prominent ASA alert type in this signature. The other two alert types are related to E23h. If a variable is allocated and ignored, this could be a possible location for a memory leak. The third alert type, M21, is an alert for using a method that is not recommended for use. A common occurrence for this alert is when the incorrect memory allocation function is called.

ASA Alert Signature 5: A Combination of the Above

The fifth ASA alert signature has a singular value of 910 and consists of 1.6% of the overall variability from the matrix of alert types. This signature is the first cluster of alert types in which we begin to see duplicates of previous alert types. The alert types are as follows:

Mistake 21: *Advisory has been issued for this function*

Security 2: *Passing untrusted input to argument*

Mistake 5: *Expression always evaluates true or false*

Warning 6: *Expression has a value not allowed for its type*

Warning 15: *then/else/loop not surrounded by braces*

Warning 5: *Operator "=" in the Boolean expression should possibly be "=="*

The alerts found in this cluster are a combination of those in the previous three clusters with one addition, W6, an ASA alert regarding a possible type mismatch between variables. Therefore, this alert signature becomes a good candidate for ending our search for ASA alert signatures. Also, with this cluster we have reached our threshold of 90% as described in previous sections [38]. Because we have reached our intended variability threshold, and because this cluster is a combination of previous clusters, we will only use ASA alert signatures 2, 3, and 4 in our analysis.

6.3.3 APPLYING THE ASA ALERT SIGNATURES

We examined the ASA alerts that were generated on the December 2006 release of the software system using the alert signatures that were previously created to identify sections of the system that may contain field failures. In this release, BEAM reported an ASA alert on 2,448 files out of a total of 5,244 files (47% of the files). Thus, our goal in applying the ASA alert signatures in this step of the methodology is to identify those files in the set of 2,448 that are most likely to contain faults that lead to field failures. We collected field failure information from December 2006 to March 2007 to determine the actual field failure-prone files. A file was considered to be failure-prone if it was involved in the reparation of at least one customer-reported field failure. The focus of our technique is to highlight areas of the code base that are the most likely areas to produce field failures based upon historical evidence regarding ASA alerts and then compare those areas to actual reported field failures. A summary of the effects of applying the ASA alert signatures can be found in Table 6.1.

Table 6.1: Summary of effects of applying ASA alert signatures

	Before Applying ASA Signatures	**After Applying ASA Signatures**
# of Files	2,448	393
% Reduction in Alerts to Examine	N/A	70.0%
% Field Failures Covered	79.6%	49.5%
% Absolute False Positive Rate Improvement	N/A	20.0%

Using the ASA alert signatures, there was a significant reduction in the number of files and ASA alerts that need to be analyzed. As mentioned, BEAM reported at least one alert in 2,448 of the 5,244 files that it was run against. After applying the ASA alert signatures, 393 of the 2448 files were identified as having alert types similar to a previous field failure. There is, however, a reduction in the number of field failures found verses checking every file that contained at least one static analysis alert. Note that our technique highlights areas of the code bases that may contain field failures based upon previous development efforts and field failure reports. The field failures that could not be identified by the ASA alert signatures do not match any previous alert patterns in reported field failures. Nearly 50% of the field failures still fall under a similar ASA alert pattern from previous releases, indicating that a large percentage of field failures come from a relatively common and consistent set of mistakes. If every file that contained a static analysis alert was examined, only 79.6% of the field failures would be detected. Research has shown that ASA tools can only find certain types of programmer errors [62] and, thus, cannot be expected to find all faults that lead to field failures.

6.4 INDUSTRIAL CASE STUDY

After we received the results of our feasibility study, we determined that there was sufficient evidence to continue in this branch of research. From our experiences with the feasibility study, we were able to refine and improve our data gathering and manipulation processes so that we could continue our investigation with a larger data set. We expanded our efforts with a large case study of our technique on six major components of an IBM software system. Our hypothesis remained the same as the feasibility study: *automated static analysis alert signatures generated from historical information through singular value decomposition can be used to identify failure-prone files*. In this section, we will discuss our research methodology and results from the case study.

6.4.1 CASE STUDY SETUP

We continued using Matlab 7.2 R2006a as our SVD tool and BEAM for generating ASA alerts. We performed our case study on six components of a large software project over six separate builds. The code base for this case study contained some of the same code from the feasibility study, but was composed of previously unexamined code. We selected these particular modules because they were primarily written in C and C++, which are two of the languages that this particular version of BEAM could analyze. The six builds span eighteen months of the development of the system. We gathered builds two to four months apart between October 2005 and March 2007. The dates for each build are provided in Table 6.2. ASA was run on each build of the software post hoc, and we gathered information on the files, alert types, and line numbers where the alerts appeared. BEAM was configured to look for all 74 alert types across the five categories for each build. ASA alerts were associated with clusters of files using our technique and the difference between the number of ASA alerts for each alert type for the two builds was calculated for generating the ASA alert signature clusters.

Table 6.2: Build dates

Build Number	Build Date
0	2005/10/31
1	2006/02/28
2	2006/06/30
3	2006/10/28
4	2006/12/11
5	2007/03/13

We also had two other research goals in mind for this case study as we examined how to best tune our technique to provide the most accurate data. To this end, we included in this study an examination of how our technique could be affected by both the amount of data used to build the cluster sets and the cluster creation threshold values. To examine the effects of different data set sizes, we created and applied signatures for every combination of build dates. The experiment data sets can be found in Table 6.3. The information in Table 6.3 outlines how we took differing time frames into account for this experiment. We created sets of ASA signatures using data that spanned two, three, four, or five builds. Further, we repeated the experiment set described in Table 6.3 three times using different cluster creation threshold values. The cluster creation threshold value determines how strongly a file or ASA alert has to be linked to other files or alerts to be considered a part of the cluster.

Table 6.3: Case study overview

Experiment Set	Signatures built using field failures between…		Signatures applied to clusters created from changes up through build…	To predict field failures between…	
	Sig. Begin Build	Sig. End Build		Prediction Begin Build	Prediction End Build
A1	0	1	1	1	3
A2	1	2	2	2	4
A3	2	3	3	3	5
A4	3	4	4	4	Present
A5	4	5	5	5	Present
B1	0	2	2	2	4
B2	1	3	3	3	5
B3	2	4	4	4	Present
B4	3	5	5	5	Present
C1	0	3	3	3	5
C2	1	4	4	4	Present
C3	2	5	5	5	Present
D1	0	4	4	4	Present
D2	1	5	5	5	Present
E1	0	5	5	5	Present

6.3.2 ASA ALERT SIGNATURES

We followed the steps outlined in Section 6.2 and highlighted in Table 6.3 for this experiment. For each experiment set, we gathered field failure information for the specified time frame as outlined in columns 2 and 3 in Table 6.3. First, we built clusters of files that tended to change together due to field failures on or before the signature end build date of the build pair. These clusters of files represented groups of files in the system that contained a specific field failure. We then built our ASA alert signatures for the build pair by performing another SVD between these clusters of files and the difference in the number of ASA alerts found in each similar file between the two builds. The difference in the number of ASA alerts was used so that correlations between field failures and changes in the density of ASA alerts can be made.

We generated a total of 74 ASA alert signatures for each experiment set because there are 74 alert types in BEAM. A graph of the singular values of the 74 ASA alert signatures for experiment set A3 can be found in Figure 6.2, which was chosen randomly from the experiment sets. The singular value graphs for all other build pairs are similar to Figures 6.1 and 6.2.

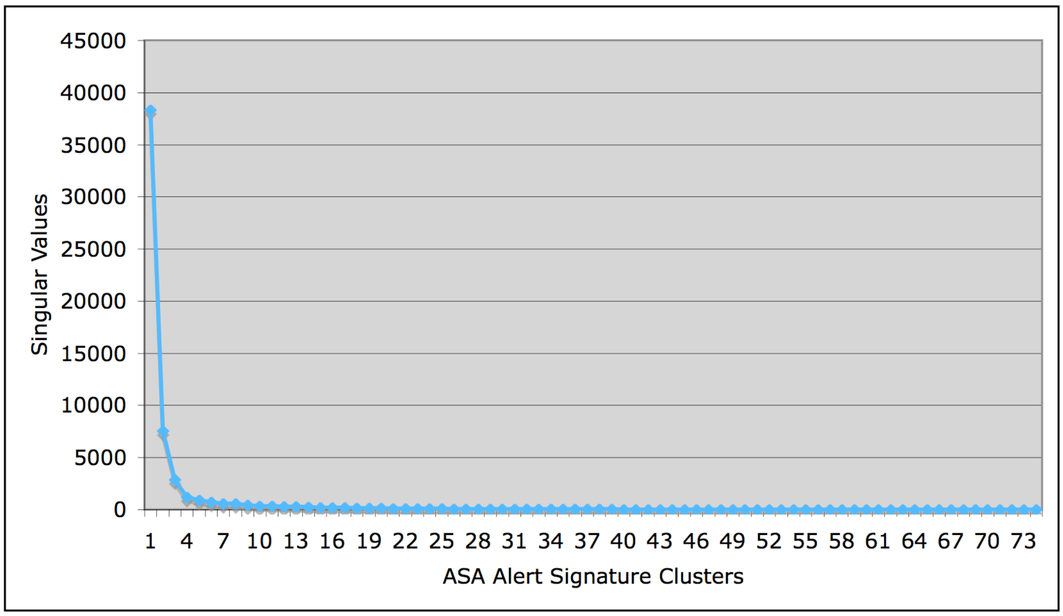


Figure 6.2: ASA alert signature cluster singular values

In a similar fashion to the feasibility study, nearly 95% of the overall variability in the original matrix is found in the first five singular values. Further, note that the first singular value is a factor of ten greater than the second. This large separation of values is an indication that the ASA alert signature associated with the first singular value would have an extremely high match rate with sets of ASA alerts. An ASA alert signature associated with the first and highest singular value consisted of only one or two ASA alerts that were pervasive in the system and would not provide any value to the set of signatures for minimizing the total number of files to examine.

Conversely, after the fifth singular value, we have reached or surpassed the 90% variability threshold in each of the fifteen experiment sets that we established for including a signature in our study. Therefore, we select the signatures associated with the second, third, fourth, and fifth singular values for each experiment set.

Across the fifteen experiment sets, we gathered four signatures for each, for a total of 60 ASA alert signatures. The 60 alert signatures were comprised of only 14 of the 74 total alert types, providing some initial indication as to which alerts were most likely to be associated with field failures. The alert types are listed below:

Error 13: *Incorrect printf argument*

Error 23h: *Heap memory leak*

Mistake 2: *Unreachable statement*

Mistake 5: *Expression always evaluates true or false*

Mistake 21: *Advisory has been issued for this function*

Warning 1: *File static variable not used*

Warning 2: *Value of variable with local scope is never used*

Warning 9: *Return value of function is not used*

Warning 12: *Cast from signed to unsigned*

Warning 14: *Possible type mismatch*

Warning 15: *then/else/loop not surrounded by braces*

Warning 16: *Function accesses same variable through two parameters*

Portability 2: *Incorrect cast*

Security 2: *Untrusted input as argument*

Each of the 60 alert signatures was comprised of two to five of the 14 identified alert types. While the ASA alert signatures fluctuated between experiment sets due to different sets of field failures and development priorities during that time frame, the two signatures discussed below appeared the most frequently between the experiment sets.

ASA Alert Signature 1: Taking a Wrong Turn

This ASA alert signature was common among 14 of the 15 experiment sets and often had the second or third highest singular value. This set of ASA alerts focuses on missing parts of an execution path. The signature consists of four alert types. The alert types are as follows:

Warning 9: *Return value of function is not used*

Mistake 5: *Expression always evaluates true or false*

Warning 14: *Possible type mismatch*

Warning 15: *then/else/loop not surrounded by braces*

These alert types are all similar in that they identify places in the code where a simple programming mistake could lead to the execution of a different path than intended. For example, if the condition of an if/else statement always evaluates to true or false, then one branch of that path is never executed. A similar problem could occur if the braces are missing around another if/else statement.

ASA Alert Signature 2: Memory Leaks

This set of three ASA alert types focuses on memory leaks and appeared in approximately half of the experiment sets. The alert types are as follows:

Error 23h: *Heap memory leak*

Warning 2: *Value of variable with local scope is never used*

Mistake 21: *Advisory has been issued for this function*

The ASA alert type for memory leaks, Error 23h, has the highest association value in this signature. The other two alert types are related to Error 23h. If a variable is allocated and ignored, this could be a possible location for a memory leak. The third alert type, Mistake 21, is an alert for using a method that is not recommended for use. A common occurrence for this alert is when the incorrect memory allocation function is called.

6.3.4 APPLYING THE SIGNATURES

Once we built our ASA alert signatures, we applied them to ASA alerts associated with clusters of files generated from all changes in the system found up until the last build date used in that experiment set to predict future field failures. Cluster sets generated from all changes show how files are likely to change together not only through field failures, but also through faults discovered through testing. We are especially interesting in these clusters since faults during testing can be an indication of post-release quality [42].

The output of the application of the ASA signatures is a set of recommended files for further investigation. These files lay along execution paths that contain similar trends of ASA alerts as found in previous field failures and thus warrant extra attention. To test how accurate our technique is at identifying failure-prone files, we examined two different metrics. We used metrics similar to those used by Canfora to evaluate his clustering algorithm [10]:

- **Precision:** We define the precision percentage as the number of recommended files that were actually changed due to field failures over the next two build cycles divided by the total number of recommended files.
- **Accuracy:** We further define the accuracy percentage as the number of failure-prone files that were recommended by our technique divided by the total number of failure-prone files.

These metrics were gathered for each experiment set for three different cluster creation threshold values. These cluster creation threshold values are reflected in the columns of result *U* matrix after performing the SVD. Higher values here indicate that the files must have a higher degree of relation through the SVD to be clustered together. A lower value allows for larger clusters of files. A summary of the results can be found in Table 6.4.

Notice that there is a distinct difference in the accuracy of our technique using different cluster creation threshold values. A lower threshold value means that a cluster will contain more files, and an ASA alert signature will contain more alert types. The lower threshold thus provides a larger data set with more specific signatures to match and leads to poorer accuracy and precision. However, with a properly tuned threshold value, the precision and accuracy of our technique improves significantly. As we experimented with cluster creation threshold values, we began with similar values used by Osinski [38] and increased the value by 0.05 for each set. We found the highest precision and accuracy percentages with a threshold of 0.25, with the percentages beginning to drop off beyond that threshold.

Table 6.4: Results summary

Exp. Set	Threshold = .15		Threshold = .2		Threshold = .25	
	Precision %	Accuracy %	Precision %	Accuracy %	Precision %	Accuracy %
A1	8.73%	9.95%	34.50%	33.15%	43.69%	25.28%
A2	6.43%	9.57%	29.17%	33.79%	31.55%	36.55%
A3	N/A	N/A	22.15%	37.63%	23.90%	40.86%
A4	8.81%	11.73%	30.12%	42.37%	30.00%	43.22%
A5	10.39%	6.58%	58.75%	37.15%	61.33%	36.36%
B1	6.43%	9.57%	29.17%	33.79%	31.55%	36.55%
B2	N/A	N/A	22.15%	37.63%	23.90%	40.86%
B3	8.81%	11.73%	30.30%	42.37%	29.82%	43.22%
B4	10.39%	6.58%	58.75%	37.15%	62.77%	33.99%
C1	N/A	N/A	22.15%	37.63%	23.90%	40.86%
C2	8.81%	11.73%	30.12%	42.37%	29.65%	43.22%
C3	10.39%	6.58%	58.75%	37.15%	61.59%	36.76%
D1	8.81%	11.73%	30.30%	42.37%	29.65%	43.22%
D2	10.39%	6.58%	58.75%	37.15%	61.59%	36.76%
E1	10.39%	6.58%	58.75%	37.15%	61.33%	36.36%
Averages	**9.07%**	**9.08%**	**38.26%**	**37.92%**	**40.42%**	**38.27%**

Another point to note from the data in Table 6.4 is that the results of the technique are relatively stable across the different experiment sets. This result is an indication that our technique can be useful with even a relatively small data set and can have similar performance to a data set that has significantly more data.

When we examined each ASA alert signature's precision and accuracy independently, we determined that 86% of the failure-prone files identified by the signatures were identified by more than one signature. Each signature did identify new failure-prone files that were not identified by the other signatures. However, there was a point of diminishing returns as new signatures with lower singular values were applied. These diminishing returns are one of the reasons for using the 90% variability threshold for selecting signature clusters.

One of the main goals of our technique is to aid developers by minimizing the number of ASA alerts that would require their attention and to highlight particular areas of the code base. The signatures created in this study only use 14 of the 74 total ASA alerts, providing a general indication of the types of alerts that would require further investigation. Using the ASA alert signatures, our technique reduced the total raw number of ASA alerts to examine by 74% and the total number of files to examine by 43%.

The set of identified potentially failure-prone files by the ASA signatures encompassed 40% of the actual failure-prone files. If a development team examined every file that contained at least one static analysis alert, only 79% of the field failures would be detected. Because our technique is based on historical data, the ASA alert signatures are not guaranteed to identify all future failure-prone files. Thus, the optimal percentage of future failure-prone files that our technique could identify is equivalent to that of all files that contain at least one static analysis alert.

6.5 COMPARISON TO OTHER TECHNIQUES

To further evaluate our technique for guiding the use of ASA alert information, we compared our results with those of another published technique using the same data set. Our technique uses historical change, fault, and failure information to generate the ASA alert signatures. Similarly, the correlation and regression approaches used by Zheng et al. [62] and Nagappan and Ball [32] also utilize historical fault information to tune the results of ASA runs. We duplicated Zheng and Nagappan's correlation analysis on our experiment code base using a Spearman correlation to determine if their techniques would be effective in with our case study and how their techniques' results would compare with our own.

Zheng and Nagappan's technique used the ASA alert density in system components to separate high- and low-quality components as determined by their pre-release fault density. Our technique, however, is focused at the file level as opposed to the component level, and as such, we only included files from six different components in our data set.

We applied Nagappan's correlation and regression technique at the file level in accordance with our technique. The initial correlation analysis is shown in Table 6.5.

Table 6.5: Summary of spearman rank correlation for 1545 files

		Total failure density	Testing failure density	Field failure density
ASA alert density	Corr. Coeff.	.232	.223	.196
	Sig.	.000	.000	.000
Total failure density	Corr. Coeff.	1	.982	.428
	Sig.	.	.000	.000
Testing failure density	Corr. Coeff.		1	.324
	Sig.		.	.000

When we apply the correlation analysis at the file level, the results are similar to previous research reports of Nagappan and Zheng. Nagappan found a statistically-significant correlation coefficient between ASA alert density and pre-release failure density of .368 in his study. In our data, we have a correlation coefficient of .232 between the same two variables. The correlation coefficient between ASA alert density and field failures was smaller than that of pre-release failures at .196. A possible reason for this is difference in correlations is that, while areas of code that contain ASA alerts that are verified and validated within rigorous testing suites may prove to be true positives and are correlated with failures, not all of these same sections of code with ASA alerts are being exercised in the field. Therefore, a correlation analysis could be more appropriate when all areas of the code base are being executed thoroughly. In SDAA, our goal in this study was to identify potential field failure-prone files. By adding empirical evidence as to what areas of the code base are being executed and where failures are being reported by customers, we can better identify potential field problems in the system.

One possible reason for the differences between the results that we received on our code base verses the efforts of Zheng et al. and Nagappan and Ball is in how the ASA alert data was pre-filtered. Zheng and Nagappan's studies used further efforts to narrow their static analysis alert set, removing a number of the false positives before performing their analysis. In Nagappan's work, he used the results of two separate (but related) static analysis tools to generate his correlations. In Zheng's work, Nortel sent their static analysis results to a third party to remove as many of the false positives as possible before going further in their process. Each study used a different set of ASA tools (we used BEAM, Nagappan used PREfast and PREfix, and Zheng used Flexelint and Klockwerk). In our study, BEAM was set to report on possible alert types at all possible severity levels.

We addressed the lack of prefiltering in our technique by utilizing the different severity levels of ASA alerts presented in BEAM. An ASA "Error" alert is indicative of a problem that would almost certainly cause a failure in the system, or even in compilation. An ASA "Warning" alert is less severe than an "Error," and could be treated similarly to a compilation warning. An ASA "Mistake" alert, is the least severe kind of alert, and it indicates areas of the code that do not conform to coding standards or good coding practice. The remaining alert types, "Security" and "Portability," report specific problems in the code base related to either security issues or compilation issues between platforms. Thus, we further analyzed how each of these warning types correlated with failures in the system, as shown in Table 6.5. We continued our investigation at the file level by continuing the correlation analysis with all the separate BEAM alert types, as shown in Table 6.6.

The highest correlation value with pre-release failure density was .241 with "Security" alert density. In contrast to the analysis at the component level, all the correlation coefficients between ASA alert types and failure densities are positive with this size data set, even if some of the coefficients are close to zero. However, again we note that the correlation coefficients for field failure density are in general lower than that of the pre-release failure density, except for the case of "Error" and "Security" alert types where the correlation coefficient rose by at least double. One explanation for this increase in the coefficient is that these particular types of errors are most likely to be failure-causing problems [8]. Thus, these alerts represent, in effect, potential faults injected into the code base during development that cannot or are not identified during the testing phase and are released to customers who then find them in the field.

Table 6.6: Further analysis of spearman rank correlation for 1545 files

		ASA "Error" density	ASA "Warn" density	ASA "Mis" density	ASA "Sec" density	ASA "Port" density	Total failure density	Testing failure density	Field failure density
ASA alert density	Corr. Coeff.	.521	.981	.829	.403	.362	.232	.223	.196
	Sig.	.000	.000	.000	.000	.000	.000	.000	.000
ASA "Error" density	Corr. Coeff.	1	.462	.436	.335	.272	.080	.071	.148
	Sig.	.	.000	.000	.000	.000	.002	.000	.000
ASA "Warn" density	Corr. Coeff.		1	.776	.372	.347	.248	.241	.210
	Sig.		.	.000	.000	.000	.000	.000	.000
ASA "Mis" density	Corr. Coeff.			1	.402	.318	.180	.178	.114
	Sig.			.	.000	.000	.000	.005	.000
ASA "Sec" density	Corr. Coeff.				1	.208	.044	.036	.209
	Sig.				.	.000	.082	.154	.000
ASA "Port" density	Corr. Coeff.					1	.113	.111	.209
	Sig.					.	.000	.000	.000
Total failure density	Corr. Coeff.						1	.982	.428
	Sig.						.	.000	.000
Testing failure density	Corr. Coeff.							1	.324
	Sig.							.	.000

We continued our comparison with Nagappan and Zheng's techniques by performing a linear regression analysis to determine the effectiveness of their technique for identifying failure-prone components as compared to SDAA. We performed a data splitting analysis to generate the regression models using two-thirds of the data to predict the failure densities of the other one-third. Regression equations were generated using the data values from the alert densities of the five BEAM alert types to predict both pre- and post-release failure densities. The results of two sets of these predictions can be found in Figures 6.3 and 6.4.

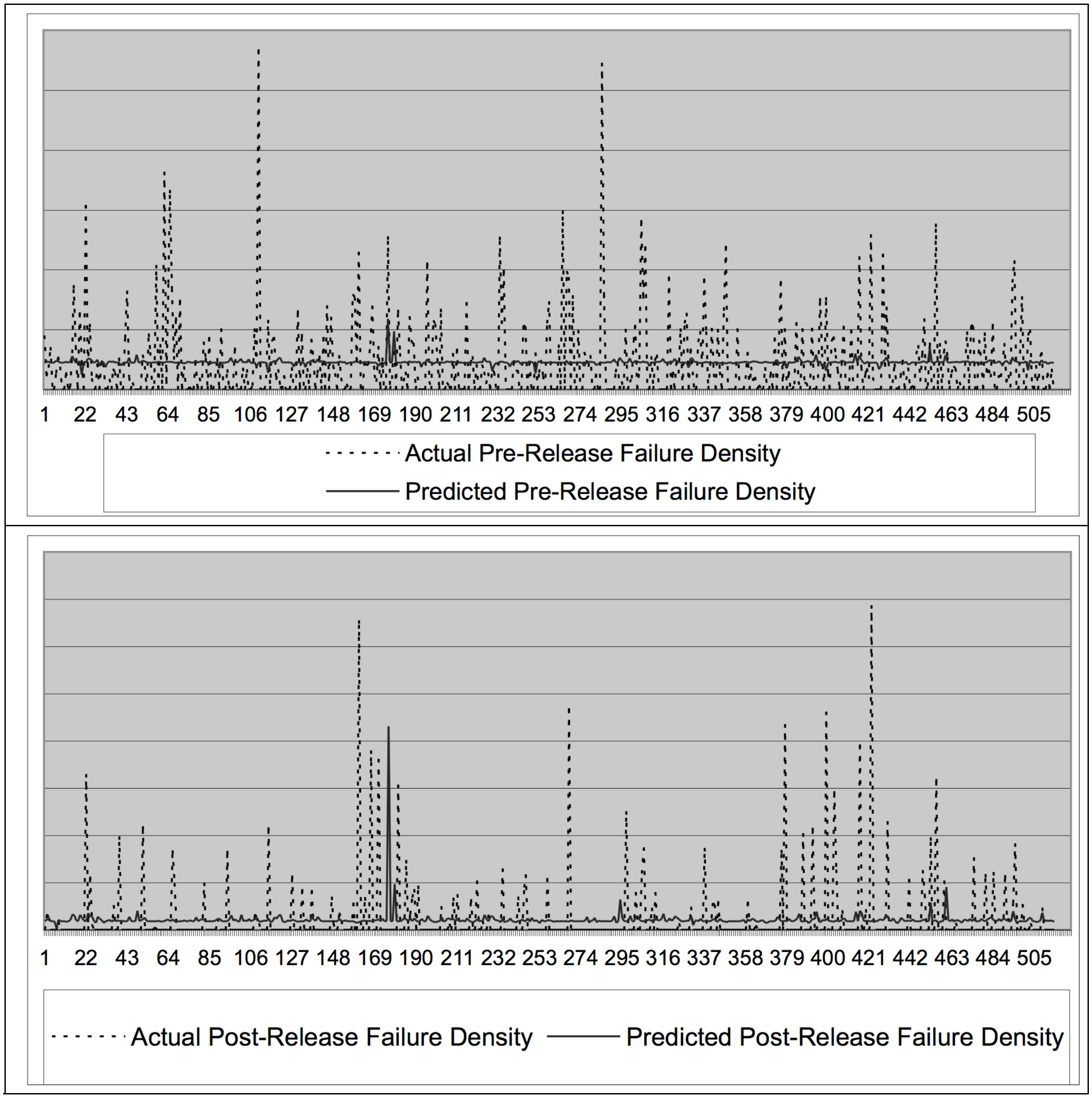


Figure 6.3: Results of using regression analysis to predict failure density (Run 1)

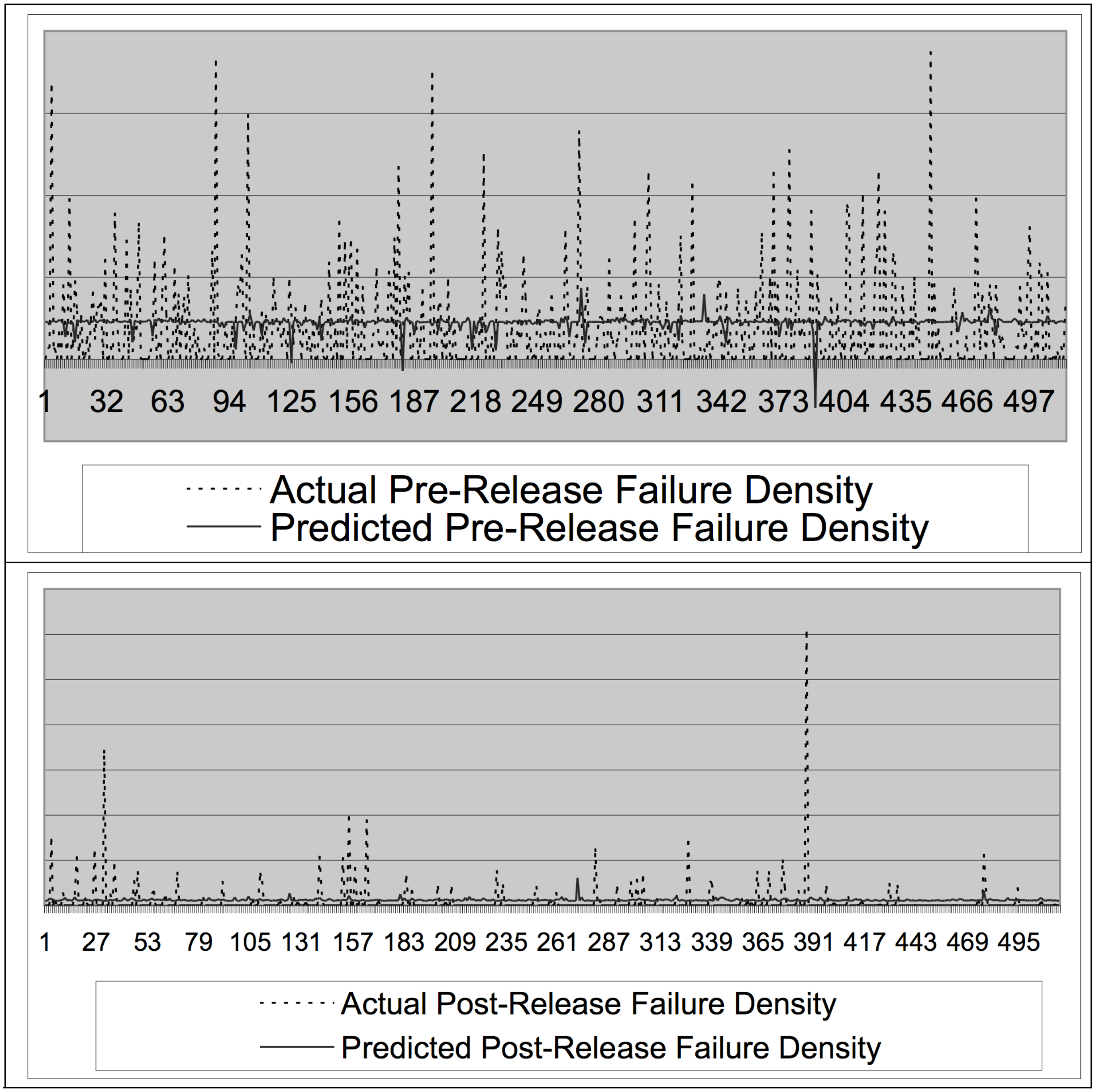


Figure 6.4: Results of using regression analysis to predict failure density (Run 2)

Figures 6.3 and 6.4 show the actual and predicted values for both pre- and post-release failure density using the regression technique as laid out by Nagappan. As is shown in Figures 6.3 and 6.4, we found that for our data set using BEAM, a regression analysis did not prove to be an accurate means to predict the failure-proneness of a file. The regression analysis predicted a relatively flat distribution of failures throughout the system, rarely spiking for files that proved to be particularly failure-prone. Using this methodology, files could not be accurately identified as failure-prone or not failure-prone. Despite similar correlations coefficients to Nagappan's study, we were not able reproduce his success in using this method for identifying failure-prone components.

There are several potential reasons for our lack of success in using a correlation/linear regression approach with our data set. First, different static analysis tools were used in the studies. There is the possibility that PREfix and PREfast are more accurate ASA tools that produce fewer false positives. However, these tools are proprietary and could not be used in our study. Second, the granularity level could also affect the ability to predict the failure-proneness of a file or module. Individual files are not as likely to contain a wide variety of ASA alert types within one file itself. Finally, the nature of the two systems could also have had an effect. Windows Server 2003 is intended to run on a single, end-user machine. However, the software we examined in our study is a distributed system that could possibly run across thousands of machines. These inherent differences in the code bases could have affected the number and types of alerts produced by the tools.

We continued our comparison with Nagappan's technique by examining his use of discriminant analysis to separate failure-prone components from non-failure-prone components. We used the same upper cut off function as used by Nagappan to determine whether a file would be considered fault- or failure-prone. When using overall fault density as the predictor in the discriminant analysis function, we find that 58% of the files are correctly classified as fault-prone or not fault-prone. We can improve the performance of the discriminant analysis to 63% if we instead use the individual densities of the three highest priority alert categories (ERROR, WARNING, and MISTAKE) of the files correctly classified. Using a discriminant analysis with these alert densities has the same true and false positive rates as our technique in its best case, as shown in previous sections. On average discriminant analysis is correct a higher percentage of the time than our technique, 56% versus 36%, as shown in Table 6.7

Table 6.7: Summary of case studies

	Average True Positive Rate	Maximum True Positive Rate
SDAA Technique	36%	63%
Regression Analysis	10%	14%
Discriminant Analysis	56%	63%

However, our technique does have one advantage in that the singular values of the static analysis signatures can used a prioritization method similar to the work described in Chapter 5 as to which identified files have a higher chance of being a true positive. A signature match with a higher singular value would be indicative of a more pervasive problem in the code base.

6.6 DEPENDANCE ON FAILURE RECORD TRACEABILITY

In the discussions above, we outlined how detailed failure tracking information regarding exactly what files were altered in response to a field failure was required. However, many projects do not record any sort of failure traceability at this level of detail. There is a possibility that the developer may make a note in the change logs that a particular failure was addressed in a revision, but records of this type, often found in open source projects, may not be detailed or standardized enough to be of reliable use in our technique. Thus the question arises as to whether our technique can have any value in an environment where we cannot directly link field failures to specific files.

After the static analysis alerts have been gathered and field failure information processed, the next step in our technique is to generate matrices that allowed us to associate groups of static analysis types with reported field failures. Then, after these static analysis signatures were identified, they could be applied to a set of static analysis alerts to help direct development effort toward the areas of code that most resembled known field failures. Without the availability of field failure traceability to individual files in the open source projects, however, we have to adjust this methodology. We began by first finding sets of static analysis alert types that appear together throughout the code base. In this way, we created our initial group of static analysis signatures. This set of signatures, however, is comprised of alert types that are related only between themselves. For example, we found that a static analysis alert regarding an empty catch block typically appeared in the same file as a static analysis alert regarding an empty finally block. While these alert types may be an indication that there is possibly a fault here, we do not know if this is a common cause of failures in this system. Thus, we need to then isolate those signatures that are most likely tied to field failures.

There are other metrics in the system that we can leverage to help us attain the goal of associating static analysis signatures to code. Research has shown that areas of high code churn in a software system are often associated with areas of high fault density in a code base [33]. Thus, we can use churn information as an approximation for defect density in our technique and use that to narrow our list of static analysis signatures. Strictly speaking, the signatures isolated in this method will be associated with areas of high code churn, which in turn are areas of the code that are likely to have a higher defect density.

During the investigation, of this version of the technique, we determined that detailed field failure records are indeed necessary for this technique to work efficiently. The signatures discovered though this manner did not associate with failure prone areas of the code base. Thus, we note the necessity for this detailed level of field failure traceability as a limitation of our technique.

6.7 CONCLUSIONS

In this chapter, we have presented a technique for combining a project's historical field failure information, change records, and static analysis alerts to generate ASA alert signatures. These alert signatures consist of groupings of ASA alert types that have been directly linked to field failures in previous builds. By applying these signatures to a current set of ASA alerts, developers can isolate specific files and alert types that historically have been associated with field failures. Our technique differentiates itself from other techniques by specifically levering historical field failure data to filter out excess static analysis alerts, leaving only those that have been shown to be linked to field failures in the past.

We performed a case study with an industrial software system at IBM to evaluate our technique. Field failure information, change records, and ASA alerts were gathered on six releases of the system over an eighteen-month period. The data from these releases were used to build ASA alert signatures that correspond to field failures found between different pairs of the six builds. We then applied these ASA alert signatures to the alert set from the latest build to predict the failure-prone files for the following two build cycles. We found that 40% of the field failures could be discovered by examining the files containing combinations of fourteen out of a possible 74 total alert types. The remaining failures were either not detected by a signature which could be an indication of a new type of error in the field, or they were on areas of the code where no static analysis alerts were detected.

We further investigated how different data set sizes and cluster creation threshold values affected the efficacy of our technique as measure by its accuracy and precision. We found that the data set size had little to no effect, showing that our technique can be used on data sets of varying sizes with comparable results. We also found that the cluster creation threshold, which controlled the size of the clusters themselves, had a significant effect on the accuracy and precision.

CHAPTER 7

CONTRIBUTIONS AND FUTURE WORK

Software developers, testers, and managers rely on various types and sources of information regarding the quality of their product. This information is used in every aspect of the development lifecycle, from requirements to coding to testing to release planning. However, gathering, processing, and interpreting this data and converting it into a form that can effectively drive development decisions can often be a costly endeavor in its own right. A great deal of information is generated throughout the development process that is often overlooked in the form of development artifacts. In this research, we have proposed and examined a framework that provides developers with guidance as to how to take various development artifacts types and generate repeatable techniques using SVD that can help answer the development decisions that must be constantly balanced during software development and maintenance.

With the aid of our industrial partners at IBM, we employed our SDAA framework to generate three techniques that can aid development decisions in impact analysis, regression test prioritization, and filtering static analysis results.

These case studies illustrated how the methods created through SDAA can provide quick and efficient guidance as to where developer effort should be directed. While the techniques may not be considered safe within their own analysis realms, the techniques distinguish themselves from current similar techniques by 1) including all files from the system and not just source code; 2) incorporating historical and empirical evidence in prioritizing developer effort; and 3) using consistently less system and development resources. An overview of these techniques and our case studies are shown in Tables 7.1, 7.2, and 7.3 below.

Table 7.1: Impact analysis case study overview

	Resource Usage	**Developer Overhead**	**Effectiveness**
Feasibility Study	5 min to gather artifacts, 2 seconds to run	Interviewed developers to determine if clusters were accurate	Clusters highlighted related areas of code through the artifacts
Industry Study	5 min to gather artifacts, 2 seconds to run	Developers must use source control system	Confirmed true positive rate over 45% with minimal overhead or effort
Open Source Study	5 min to gather artifacts, 2 seconds to run	Developers must use source control system; found that continuous integration of smaller revisions improves performance	Confirmed true positive rate over 65% with projects with numerous revisions of smaller sizes

Table 7.2: Regression test prioritization case study overview

	Resource Usage	Developer Overhead	Effectiveness
Industry Study	10 min to gather artifacts, 4 seconds to run	Test case traceability to files exercised is required	Confirmed true positive rate over 40%, higher probability of true positive at top of prioritization
Comparison Study	Significantly less that comparable techniques	Overhead less on average than comparable techniques	Lower resource costs overall with comparable performance to non-safe methods

Table 7.3: Static analysis signatures case study overview

	Resource Usage	Developer Overhead	Effectiveness
Feasibility Study	3 hours to run static analysis tool, 15 min to gather artifacts, 10 seconds to run	Field failure traceability to files is required	70% reduction in alerts to examine; 50% of all field failures related to an alert signature
Industry Study	3 hours to run static analysis tool, 15 min to gather artifacts, 10 seconds to run	Field failure traceability to files is required	74% reduction in alerts to examine; up to 60% of all field failures related to an alert signature
Comparison Study	Depends on the static analysis tool chosen; other studies performed external reviews of static analysis alerts which increases overall cost and time	Failure traceability is required to files is required for our technique; however, our technique can provide more information on the nature of failures due to the signatures as opposed to raw alert density	Our technique showed some improvement over regression analysis in certain cases; however, on average, a discriminant analysis performed better than our technique

Our results support the theory proposed in Chapter 1:

Decision support for software development decisions is more resource efficient when based upon relationships between software artifacts illuminated via a singular value decomposition of actual usage information than when based upon semantic techniques.

Our motivation in this research is to provide decision support for industry in instances where resource efficiency and overall time are critical. Through our studies, we have shown that software development artifacts can be a viable means to help guide development decisions by leveraging the artifacts to illuminate hidden relationships in the system through SVD. Our techniques and overall methodology would be appropriate for use with teams that have less overall resources for the development of non-critical software systems. We recognize the trade off between inclusiveness and precision with the resources required to execute the technique.

The techniques in this research only illustrate a small number of possible artifacts and decisions that could be considered using SDAA. In our future work, we plan to apply our technique to more general program comprehension techniques. Further, more types of analyses could be performed with various other sets of software development artifacts. For example, a similar analysis to the previous examples could be performed with only field failures as opposed to the entire set of reported faults and failures. Comparing the results from an analysis using changes from faults found in testing and the results from an analysis using changes from field failures would provide insight into what files were being tested internally versus what files were being changed due to customer-reported failures. If parts of the two analyses resulted in completely orthogonal association clusters from each other, then there is a possibility of a lack of testing coverage in the areas identified by the field failure analysis. We will investigate the use of various subsets of change information along with other software development artifacts. By creating association clusters based on various types of artifacts, traceabilities could be achieved between field failures, test cases, and the identified change association clusters that they all touch. Through these traceabilities, developers and managers could have a detailed view of the risky areas of their system at any given point, helping them to make development and release decisions.

REFERENCES

[1] Ambler, S. and Jeffries, R., *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*: John Wiley & Sons, 2001.

[2] ANSI/IEEE, "IEEE Standard Glossary of of Software Engineering Terminology, Standard 729," 1983.

[3] Arnold, R. and Bohner, S., "Impact Analysis - Towards A Framework for Comparison," Conference on Software Maintenance, Montreal, Canada, 1993, pp. 292-301.

[4] Ball, T., Kim, J., Potter, A., and Siy, H., "If your version control system could talk," Workshop on Process Modeling and Empirical Studies of Software Engineering, 1997.

[5] Beck, K., *Extreme Programming Explained, Embrace Change*: Addison Wesley, 2000.

[6] Beyer, D. and Noack, A., "Clustering Software Artifacts Based on Frequent Common Changes," 13th IEEE International Workshop on Program Comprehension, St. Louis, MO, 2005, pp. 259-268.

[7] Bohner, S., ""A Graph Traceability Approach to Software Change Impact Analysis"," *Department of Computer Science*, vol. PhD. Fairfax, VA: George Mason University, 1995.

[8] Brand, D., "Error Detection by Data Flow Analysis Restricted to Executable Paths," IBM T. J. Watson Research Center, Yorktown Heights, NY, USA RC21484, May 4 1999.

[9] Breech, B., Tegtmeyer, M., and Pollock, L., "A Comparison of Online and Dynamic Impact Analysis Algorithms," European Conference on Software Maintenance and Reengineering, Manchester, UK, 2005, pp. 143-152.

[10] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories," International Symposium on Software Metrics, Coma, Italy, 2005, pp. 9-18.

[11] Cok, D. R. and Kiniry, J. R., "ESC/Java2: Uniting ESC/Java and JML: Progress and Issues in Building and Using ESC/Java2, Including a Case Study Involving the Use of the Tool to Verify Portions of an Internet Voting Tally System," International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Marseille, France, 2004, pp. 108-128.

[12] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C., and Zheng, H., "Bandera: Extracting Finite-state Models from Java Source Code," International Conference on Software Engineering, Limerick, Ireland, 2000, pp. 439-448.

[13] Demmel, J., *Applied Numerical Linear Algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

[14] Gall, H., Jazayeri, M., and Krajewski, J., "CVS Release History Data for Detecting Logical Couplings," Sixth International Workshop on Principles of Software Evolution, 2003.

[15] Hannay, J. E., Sjøberg, D. I. K., and Dybå, T., "A Systematic Review of Theory Use in Software Engineering Experiments," *IEEE Transactions on Software Engineering*, vol. 33, pp. 87-107, 2007.

[16] Heckman, S. and Williams, L., "Automated Adaptive Ranking and Filtering of Static Analysis Alerts," International Symposium on Software Reliability Engineering, Raleigh, NC, 2006.

[17] Hovemeyer, D. and Pugh, W., "Finding Bugs is Easy," *SIGPLAN Notices*, vol. 39, 2004.

[18] Huang, L. and Song, Y.-T., "Dynamic Impact Analysis Using Execution Profile Tracing," International Conference on Software Engineering Research, Management, and Applications, 2006, pp. 237-244.

[19] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[20] Jalote, P., *Software Project Management in Practice*. New York: Addison Wesley Professional, 2002.

[21] Karlsson, J. and Ryan, K., "A Cost-Value Approach for Prioritizing Requirements," *IEEE Software*, vol. 14, pp. 67-74, Sep-Oct 1997.

[22] Khoshgoftaar, T. M., Allen, E.B., Kalaichelvan, K.S., Goel, N., Hudepohl, J.P., Mayrand, J., "Detection of fault-prone program modules in a very large

telecommunications system," International Symposium Software Reliability Engineering, 1995, pp. 24-33.

[23] Khoshgoftaar, T. M., Munson, J.C.,, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 253-261, 1990.

[24] Kroll, P. and Kruchten, P., *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Boston: Addison Wesley, 2003.

[25] Kruchten, P., *The Rational Unified Process: An Introduction*, Third ed. Boston: Addison Wesley, 2004.

[26] Law, J. and Rothermel, G., "Whole Program Path-Based Dynamic Impact Analysis," International Conference on Software Engineering, Portland, OR, 2003, pp. 308-318.

[27] Livshits, B. and Zimmermann, T., "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," European Software Engineering Conference and Symposion on the Foundations on Software Engineering, Lisbon, Portugal, 2005.

[28] Musa, J., "Theory of Software Reliability and its Applications," *IEEE Transactions on Software Engineering*, pp. 312-327, 1975.

[29] Musa, J., *Software Reliability Engineering*. New York, NY: McGraw-Hill, 1999.

[30] Musa, J. D., "Operational Profiles in Software Reliability Engineering," *IEEE Software*, vol. 10, pp. 14-32, 1993.

[31] Nagappan, N., "A Software Testing and Reliability Early Warning (STREW) Metric Suite," *Department of Computer Science*, vol. PhD. Raleigh, NC: North Carolina State University, 2005.

[32] Nagappan, N. and Ball, T., "Static Analysis Tools as Early Indicator of Pre-Release Defect Density," International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 580-586.

[33] Nagappan, N. and Ball, T., "Use of Relative Code Churn Metrics to Predict System Defect Density," International Conference on Software Engineering, St. Louis, MO, 2005, pp. 235-244.

[34] Nagappan, N., Williams, L., Vouk, M., Hudepohl, J., and Snipes, W., "A Preliminary Investigation of Automated Software Inspection," IEEE International Symposium on Software Reliability Engineering (ISSRE), St. Malo, France, 2004, pp. 429-439.

[35] Nagappan, N., Williams, L., Vouk M.A., ""Good Enough" Software Reliability Estimation Plug-in for Eclipse," IBM-ETX Workshop, in conjunction with OOPSLA 2003, 2003, pp. 36-40.

[36] Orso, A., Apiwattanapong, T., and Harrold, M. J., "Leveraging field data for impact analysis and regression testing," Symposium on the Foundations of Software Engineering, Helsinki, Finland, 2003, pp. 128-137.

[37] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., and Harrold, M. J., "An Empirical Comparison of Dynamic Impact Analysis Algorithms," International Conference on Software Engineering, Scotland, 2004.

[38] Osinski, S., Stefanowski, J., and Weiss, D., "Lingo: Search Results Clustering Algorithm Based on Singular Value Decomposition," Advances in Soft Computing, Intelligent Information Processing and Web Mining, Zakopane, Poland, 2004, pp. 359-368.

[39] Ostrand, T., Weyuker, E. J., and Bell, R., "Where the bugs are," International Symposium on Software Testing and Analysis, Boston, MA, 2004, pp. 86-96.

[40] ReasoningInc., "Automated Software Inspection: A New Approach to Increase Software Quality and Productivity," 2003.

[41] Ren, X., Shah, F., Tip, F., Ryder, B., and Chesley, O., "Chianti: a tool for change impact analysis of Java programs," Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, 2004, pp. 432-448.

[42] Rivers, A. T., Vouk, M.A.,, "Resource-Constrained Non-Operational Testing of Software," International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998, pp. 154-163.

[43] Rothermel, G. and Harrold, M., "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, pp. 529-551, 1996.

[44] Rothermel, G. and Harrold, M., "Selecting Tests and Identifying Coverage Requirements for Modified Software," ACM International Symposium on Software Testing and Analysis, Seattle, WA, August 1994, pp. 169-184.

[45] Rothermel, G. and Harrold, M. J., "A Safe, Efficient Algorithm for Regression Test Selection," International Conference on Software Maintenence, Montreal, Canada, 1993, pp. 358-367.

[46] Rothermel, G. and Harrold, M. J., "A Framework for Evaluating Regression Test Selection Techniques," International Conference on Software Engineering, Sorrento, Italy, 1994, pp. 201-210.

[47] Rothermel, G. and Harrold, M. J., "Empirical Studies of a Safe Regression Test Selection Technique," *IEEE Transactions on Software Engineering*, vol. 24, pp. 401-418, 1998.

[48] Rothermel, G., Untch, R., Chu, C., and Harrold, M., "Test Case Prioritization," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929-948, October, 2001.

[49] Rothermel, G., Untch, R., Chu, C., and Harrold, M., "Test Case Prioritization: An Empirical Study," International Conference on Software Maintenance, Oxford, UK, September 1999, pp. 179 - 188.

[50] Rothermel, G., Untch, R. H., and Harrold, M. J., "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929-948, 2001.

[51] Rutar, N., Almazan, C. B., and Foster, J. S., "A Comparison of Bug Finding Tools for Java," 15th IEEE International Symposium on Software Reliability Engineering, St.Malo, Bretagne, France, 2004, pp. 245-256.

[52] Sherriff, M., Heckman, S., Lake, M., and Williams, L., "Identifying Fault-Prone Files Using Static Analysis Alerts Through Singular Value Decomposition," CASCON, Toronto, Canada, 2007, pp. To appear.

[53] Sherriff, M., Heckman, S., Lake, M., and Williams, L., "Using Groupings of Static Analysis Alerts to Identify Files Likely to Contain Field Failures," Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 2007, pp. To appear.

[54] Sherriff, M., Lake, M., and Williams, L., "Empirical Impact Analysis using Singular Value Decomposition," North Carolina State University, Raleigh, NC CSC-TR-2007-13, April 29 2007.

[55] Sherriff, M., Lake, M., and Williams, L., "Empirically-Based Impact Analysis for Regression Test Selection," North Carolina State University CSC-TR-2007-11, March 20 2007.

[56] Sherriff, M., Lake, M., and Williams, L., "Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records," International Symposium on Software Reliability Engineering, Trollhättan, Sweden, 2007, pp. To appear.

[57] Srikanth, H., "Requirements-Based Test Case Prioritization," Doctoral Symposium in International Conference of Software Engineering, St. Louis, MO, 2005.

[58] Tip, F., "A survey of program slicing tecniques," *Journal of Programming Languages*, vol. 3, pp. 121-189, 1995.

[59] von Knethen, A. and Grund, M., "QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces," International Conference on Software Maintenance, 2003, pp. 246-255.

[60] Will, T., "Introduction to the Singular Value Decomposition," vol. 2006: UW-La Crosse, 1999.

[61] Ying, A., Murphy, G., Ng, R., and Chu-Carroll, M., "Prediction Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering*, vol. 30, pp. 574-586, 2004.

[62] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J., and Vouk, M., "On the Value of Static Analysis for Fault Detection in Software," *IEEE Transactions on Software Engineering*, vol. 32, pp. 240-253, 2006.

[63] Zimmermann, T., Diehl, S., and Zeller, A., "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Engineering*, vol. 31, 2005.

[64] Zitser, M., Lippmann, R., and Leek, T., "Testing static analysis tools using exploitable buffer overflows from open source code," International Symposium on Foundations of Software Engineering, Newport Beach, CA, 2004, pp. 97-106.

Appendix

APPENDIX A

Example cluster creation Matlab script

```
function T = opensource(project_name, twothirdsize, svd_depth,
currentrun);

      % Open MySQL connection
      mysql( 'open', ‘localhost’, 'mssherri', ‘password’)
      mysql('use', 'opensource')
      query = ['select max(localId) from ' project_name '_files']

      num_of_files = mysql(query);

      clusters_table = strcat(project_name, '_clusters_',
int2str(currentrun));

      query = ['CREATE TABLE ' clusters_table ' (releaseName varchar(25)
NOT NULL, clusterId int(11) NOT NULL , singularValue double NOT NULL, KEY
releaseName (releaseName),KEY clusterId (clusterId))']
      mysql(query);

      fileclusters_table = strcat(project_name, '_fileclusters_',
int2str(currentrun));

      query = ['CREATE TABLE ' fileclusters_table ' (releaseName
varchar(25) NOT NULL, localId int(11) NOT NULL , clusterId int(11) NOT
NULL ,strength double NOT NULL, KEY localId (localId),KEY clusterId
(clusterId))']
      mysql(query);

      runtwothirds_table = strcat(project_name, '_twothirds_',
int2str(currentrun));

      query = ['CREATE TABLE ' runtwothirds_table ' (revision int(11) NOT
NULL) ']
      mysql(query);

      runonethird_table = strcat(project_name, '_onethird_',
int2str(currentrun));

      query = ['CREATE TABLE ' runonethird_table ' (revision int(11) NOT
NULL) ']
      mysql(query);
```

```
      query = ['insert into ' runtwothirds_table ' (select revision from '
project_name '_revisions order by rand() limit ' int2str(twothirdsize)
')']
      mysql(query);


      query = ['insert into ' runonethird_table ' (select distinct
r.revision from ' project_name '_revisions r where r.revision not in
(select revision from ' runtwothirds_table '))']
      mysql(query);

      % Generate M coordinate set for testing defects

      query = ['select r1.localId, r2.localId, total from (select r1.path
as x, r2.path as y, count(*) as total from (select distinct c.revision,
c.path from ' project_name ' c, ' runtwothirds_table ' t where c.revision
= t.revision) as r1, (select distinct c.revision, c.path from '
project_name ' c, ' runtwothirds_table ' t where c.revision = t.revision)
as r2 where r1.revision = r2.revision group by x, y order by count(*)
desc, x, y) as m, ' project_name '_files r1, ' project_name '_files r2
where r1.path = x and r2.path = y']

      [M_coord1, M_coord2, M_coord3] = mysql(query);
      M_coord = [M_coord1 M_coord2 M_coord3];

      % Convert M_coord matrix into sparse M matrix
      M = sparse(num_of_files, num_of_files);

      for i=1:length(M_coord)
            M(M_coord(i,1), M_coord(i,2)) = M_coord(i,3);
      end


      % Compute the SVD of M
      [U, S, V] = svds(M, svd_depth);

      index_i = 1;
      index_j = 1;

      for i=1:size(U,2)
            index_i = i
            query = ['insert into ' clusters_table ' values('''
project_name ''', ' int2str(i) ', ' num2str(S(i,i)) ')']
            mysql(query);
            for j=1:size(U,1)
                  if U(j,i) > .05
                        query = ['insert into ' fileclusters_table '
values(''' project_name ''', ' int2str(j) ', ' int2str(i) ', '
num2str(U(j,i)) ')']

                        mysql(query);
                  end
            end
      end
```