

Empirical Software Change Impact Analysis using Singular Value Decomposition

Mark Sherriff and Laurie Williams
University of Virginia, North Carolina State University
sherriff@cs.virginia.edu, williams@csc.ncsu.edu

Abstract

Verification and validation techniques often generate various forms of software development artifacts. Change records created from verification and validation efforts show how files in the system tend to change together in response to fixes for identified faults and failures. We propose a methodology for determining the impact of a new system modification by analyzing software change records through singular value decomposition. This methodology generates clusters of files that historically tend to change together to address faults and failures found in the code base. We performed a post hoc case study using this technique on five open source software systems. We determined that our technique was effective in identifying impacted files in a system from an introduced change when the developers tended to make small, targeted updates to the source system regularly. We further compared our technique against two other impact analysis techniques (`PathImpact` and `CoverageImpact`) and found that our technique provided comparable results, while also identifying non-source files that could be impacted by the change.

1. Introduction

During development, testing, and maintenance, a modification made to a system can often have side effects. 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 the system modifications through impact analysis techniques [1]. However, current impact analysis techniques that utilize call graphs, dynamic executions of the system, or static code analysis often do not include files that are not part of the source code, such as media files, help files, and configuration files [5, 8, 13, 16]. Additionally, current impact analysis

techniques based upon semantic analysis may 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 require more effort to determine exactly which areas of the system have the highest risk of containing a latent fault [12].

Our research objective 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 and prioritize potentially-affected areas of a system modification based on risk*. 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. Analyzing revisions can provide an indication as to how files interact with one another to perform a system modification [2, 3]. We define a *revision* as any set of files changed together in a system modification for the specific purpose of repairing a fault or implementing a new feature in the system.

Our methodology involves the generation of association clusters from a set of change records and then leveraging those clusters to guide impact analysis. The data from change records are compiled into a matrix that portrays the historically-based change relationship between sets of files. A singular value decomposition (SVD) [6] is performed on the matrix to generate the association clusters. The results of the SVD can then be utilized to identify the potential effects of a change. *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.*

To determine the efficacy of our technique, we examined the size of the impact sets and the accuracy of our impact technique with five open source Java projects. Further, we compared our technique's performance against two established impact analysis techniques, `PathImpact` and `CoverageImpact`.

In Section 2, we will discuss relevant background and related work. Section 3 outlines the steps of our technique, while Section 4 describes our open source case study. Section 5 provides a comparison between our technique and PathImpact and CoverageImpact. Section 6 discusses our conclusions from this work.

2. Background

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 [1]. 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 [8, 13]. 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 an instrumented execution of the system using a comprehensive test suite. CoverageImpact utilizes the coverage information of each system execution with program slicing [21] 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 [21], assuming that the tests are reliable and/or the execution of the test cases extensively uses the system. If the system has little testing, or if that testing is inadequate, the efficacy of these techniques will be severely impaired.

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 [8] or the semantics of the source code itself [1, 16, 22, 25]. Orso demonstrated that static impact analysis techniques are “generally imprecise and tend to overestimate the effect of a change” [12, 13]. 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 [8, 12].

Our technique is a static impact analysis technique and addresses the concerns expressed by Orso and Huang. 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 [2, 3, 7, 11, 24, 25]. Zimmerman et al. [25] 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. 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 [25]. Further, there is also work on creating a “fault architecture” by von Mayrhofer, in which fault-prone components are identified by past defect records [23]. Our technique is similar to these methods 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.

We have performed other work using SVD to gather information from software development artifacts. Using change records and static analysis information, we prioritized static analysis alerts by forming association clusters that linked types of alerts to areas of the system that contained failures previously [17-19]. In an industrial case study using a system from IBM, we determined that using SVD as a method for prioritizing alerts reduced developer effort in analyzing static analysis alerts by 60%. We have also applied our SVD methodology to regression test prioritization [20]. Using the data from our impact analysis technique along with test coverage data through change records, we can drive test selection based upon empirical, historical evidence as to which areas of the system have had the most severe problems.

3. Empirical software change impact analysis

Our technique 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 describe the seven steps of our technique with regards to the goal of impact analysis, which includes deriving the association clusters from change records and interpreting the results of the analysis. An overview of the algorithm is presented in Figure 1.

```

1 Create matrix M where the values in
the matrix indicate the number of times
two files have changed together.
2 [U, S, V] = svd(M);
3 for i:size of U
4     Gather cluster i information
5     for j:size of U
6         if |U(j, i)| > threshold
7             Gather element of cluster i
8     end
9 end
10 end
11
12 X = list of files under change
13 Compare contents of X with each
cluster to find exact matches
14 if perfect matches found
15 return matched files
16 Search clusters for any files from X
for any cluster match
17 return any matched files

```

Figure 1. Psuedo-code for SVD-based impact analysis

3.1. Step 1: Gather data and build M matrix

The first step is to generate an analysis matrix 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 revision 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.

We have built an example analysis matrix M , shown below in Equation 1. For example, matrix M shows that File 2 has appeared in a revision 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.

$$M = \begin{bmatrix} F1 & F2 & F3 & F4 & F5 \\ F1 & 25 & 10 & 0 & 0 & 0 \\ F2 & 10 & 31 & 21 & 0 & 0 \\ F3 & 0 & 21 & 24 & 0 & 0 \\ F4 & 0 & 0 & 0 & 15 & 12 \\ F5 & 0 & 0 & 0 & 12 & 17 \end{bmatrix} \quad (1)$$

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.

3.2. Step 2: 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 matrix M . The strength of the association is determined by the frequency of time the files changed together. A SVD of matrix M provides the following matrices, shown in Equations 2 and 3:

$$U = V = \begin{bmatrix} -.29 & 0 & .9 & .31 & 0 \\ -.76 & 0 & -.02 & -.56 & 0 \\ -.59 & 0 & -.43 & .69 & 0 \\ 0 & -.68 & 0 & 0 & -.74 \\ 0 & -.74 & 0 & 0 & .68 \end{bmatrix} \quad (2)$$

$$S = \begin{bmatrix} 51.1 & 0 & 0 & 0 & 0 \\ 0 & 28.4 & 0 & 0 & 0 \\ 0 & 0 & 24.8 & 0 & 0 \\ 0 & 0 & 0 & 4.1 & 0 \\ 0 & 0 & 0 & 0 & 3.9 \end{bmatrix} \quad (3)$$

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, because M is a symmetric matrix.

A cluster's strength, represented by the size of the singular value from matrix S coupled with it, indicates the amount of variability that the association cluster provides to the original analysis matrix. The relative variability indicated by the magnitude of the singular value provides direct evidence as to how important that cluster is to the overall system. The more important the cluster is, the more risk that is associated with that

cluster, as evidenced by its velocity of change in previous system modifications.

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 . 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. 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.

3.3. Step 3: 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. 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. 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.

3.4. Step 4: Applying the clusters

Using the U and S matrices generated from our technique, we can determine the possible impact that a new revision might have on the system. We can compare the associations gathered from the U matrix with a new revision. If all the files in a new revision 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 revision 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 [14]. 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 [14].

3.5. Limitations

The association clusters are based upon the change records that may or may not be accurate because of 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. 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 that contains numerous non-executable files. In our previous industrial research, we analyzed systems using an SVD-based technique where 25% of the files were non-source files [18, 19]. 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. Open source project case study

In this section, we discuss the application of our technique to five open source projects obtained from SourceForge (<http://www.sourceforge.net>), an open source change management and resource site.

4.1. Case study setup

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, showing that the project is used;
- development stage 4 or higher, showing that the project is stable; and,
- use of Subversion change management software to gather change records.

Using the criteria above, we selected the five open source projects outlined in Table 1.

Table 1: Selected open source projects

Project	Type of software	# Files	# Revs
Abbot ¹	Java GUI testing framework	6282	2460
Grinder ²	Java load testing framework	3260	3520
HTTPUnit ³	Web site unit testing	697	779
jFreeChart ⁴	Java graphical/charting library	1245	135
StatSVN ⁵	Gathers Subversion metrics	595	342

We used Matlab 7.2 R2006a as our SVD tool. 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 each open source project in this investigation to create our data sets. We began by randomly selecting two-thirds of the revisions 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 revisions was then used as our “future set,” which would simulate incoming revisions made to perform a system modification. We performed this data splitting exercise ten separate times for each project.

¹ <http://abbot.sourceforge.net/>

² <http://grinder.sourceforge.net/>

³ <http://httpunit.sourceforge.net/>

⁴ <http://www.jfree.org/jfreechart/>

⁵ <http://www.statsvn.org/>

The impact analysis techniques used by Orso et al. [12] and Law and Rothermel [10] 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. Similar to our previous work [17], 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.

4.2 Case study results

We applied the three impact methods to each of the ten random data splits for each of the five open source projects. The results of the three impact methods are shown in Table 2. We observed a decrease in impact size when we looked at files identified as being related to changed files through the association clusters. We then examined the efficacy of Impact Method 1 by

investigating its true positive rate. The results of this examination are shown in Table 3.

Table 2: 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

Note that the efficacy of our technique in terms of true positive rate 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 files), 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 than the other projects. The larger average revision size along with fewer total revisions could be an indication of more development effort taking place before the developer checks in their work Subversion. Note that in these instances the developer may have implemented more than one feature change to the system, 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. 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

Table 3: Investigation summary

Project	Confirmed Positive Rate	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
HTTPUnit		41.1%	5.9%	5.2	4.4
jFreeChart		19.5%	0.5%	14.8	9.4
StatSVN		35.5%	5.3%	5.4	8.5

minimized regardless of the commercial nature of the project.

5. Comparison to other 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 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 [4, 13].

We believe the advantage to using our technique is three-fold. First, we believe that our technique requires fewer system resources and less time to execute than other techniques due to the use of change records as opposed to dynamic path or coverage information. 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 [15], 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.

5.1. 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 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 [13] so that the data is not 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 [4]. 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 paper, 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 $O(F*C)$.

Table 5: 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.
Our 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

The three algorithms' steps and algorithmic complexities are shown in Tables 4 and 5. 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. 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 [4]. 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, five of the systems selected could not use PathImpact effectively due to resource constraints that stemmed from the size of the system [4]. 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).

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

Table 4: 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
Our Technique	SVD ($O(F \log F)$)	$O(F*C)$	Occasional re-run of SVD

`PathImpact`. `CoverageImpact`, however, also could not be run effectively on `gcc` due to problems with creating static slices of the system [4].

5.2. Non-source files

In our 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, similar non-source 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.

6. Conclusions

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 SVD with 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 [9]. Further, our technique utilizes historical, empirical evidence as to areas of the system currently under change to highlight files that are most likely to change together. By utilizing empirical change data, our technique can effectively prioritize which areas of the system require more developer attention due to system modifications, thus helping to mitigate the risks associated with software revisions.

To examine the efficacy of our technique, we performed a case study with five open source projects. 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 `CoverageImpact` 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.

7. Acknowledgments

We would sincerely like to thank Dr. Michael Lake at IBM Corp. for his input into this work. Partial funding for this work was provided by the National Science Foundation.

8. References

- [1] Arnold, R. and Bohner, S., "Impact Analysis - Towards A Framework for Comparison," *Proceedings of the Conference on Software Maintenance*, Montreal, Canada, 1993, pp. 292-301.
- [2] Ball, T., Kim, J., Potter, A., and Siy, H., "If your version control system could talk," *Proceedings of the Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.
- [3] Beyer, D. and Noack, A., "Clustering Software Artifacts Based on Frequent Common Changes," *Proceedings of the 13th IEEE International Workshop on Program Comprehension*, St. Louis, MO, May 15-16, 2005, pp. 259-268.
- [4] Breech, B., Tegtmeyer, M., and Pollock, L., "A Comparison of Online and Dynamic Impact Analysis Algorithms," *Proceedings of the European Conference on Software Maintenance and Reengineering*, Manchester, UK, March 21-23, 2005, pp. 143-152.
- [5] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories," *Proceedings of the International Symposium on Software Metrics*, Como, Italy, September 19-22, 2005, pp. 9-18.

- [6] Demmel, J., *Applied Numerical Linear Algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 1997.
- [7] Gall, H., Jazayeri, M., and Krajewski, J., "CVS Release History Data for Detecting Logical Couplings," *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, 2003.
- [8] Huang, L. and Song, Y.-T., "Dynamic Impact Analysis Using Execution Profile Tracing," *Proceedings of the International Conference on Software Engineering Research, Management, and Applications*, Aug 9-11, 2006, pp. 237-244.
- [9] Jalote, P., *Software Project Management in Practice*. New York: Addison Wesley Professional, 2002.
- [10] Law, J. and Rothermel, G., "Whole Program Path-Based Dynamic Impact Analysis," *Proceedings of the International Conference on Software Engineering*, Portland, OR, May 3-10, 2003, pp. 308-318.
- [11] Livshits, B. and Zimmermann, T., "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, September 5-9, 2005.
- [12] Orso, A., Apiwattanapong, T., and Harrold, M. J., "Leveraging field data for impact analysis and regression testing," *Proceedings of the Symposium on the Foundations of Software Engineering*, Helsinki, Finland, September 1-5, 2003, pp. 128-137.
- [13] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., and Harrold, M. J., "An Empirical Comparison of Dynamic Impact Analysis Algorithms," *Proceedings of the International Conference on Software Engineering*, Scotland, 2004.
- [14] Osinski, S., Stefanowski, J., and Weiss, D., "Lingo: Search Results Clustering Algorithm Based on Singular Value Decomposition," *Proceedings of the Advances in Soft Computing, Intelligent Information Processing and Web Mining*, Zakopane, Poland, 2004, pp. 359-368.
- [15] Ostrand, T., Weyuker, E. J., and Bell, R., "Where the bugs are," *Proceedings of the International Symposium on Software Testing and Analysis*, Boston, MA, 2004, pp. 86-96.
- [16] Ren, X., Shah, F., Tip, F., Ryder, B., and Chesley, O., "Chianti: a tool for change impact analysis of Java programs," *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004, 2004, pp. 432-448.
- [17] Sherriff, M., *Analyzing Software Artifacts Through Singular Value Decomposition to Guide Development Decisions*, Diss, Department of Computer Science, North Carolina State University, 2007.
- [18] Sherriff, M., Heckman, S., Lake, M., and Williams, L., "Identifying Fault-Prone Files Using Static Analysis Alerts Through Singular Value Decomposition," *Proceedings of the CASCON*, Toronto, Canada, Oct 13-15, 2007, p. To appear.
- [19] Sherriff, M., Heckman, S., Lake, M., and Williams, L., "Using Groupings of Static Analysis Alerts to Identify Files Likely to Contain Field Failures," *Proceedings of the Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sept 7-10, 2007, p. To appear.
- [20] Sherriff, M., Lake, M., and Williams, L., "Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records," *Proceedings of the International Symposium on Software Reliability Engineering*, Trollhättan, Sweden, Nov 7-10, 2007, p. To appear.
- [21] Tip, F., "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 1, pp. 121-189, 1995.
- [22] von Knethen, A. and Grund, M., "QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces," *Proceedings of the International Conference on Software Maintenance*, September 22-26, 2003, pp. 246-255.
- [23] von Mayrehaurser, A., Wang, J., Ohlsson, M., and C., W., "Deriving a Fault Architecture from Defect History," *Proceedings of the International Symposium on Software Reliability Engineering*, Boca Raton, Florida, Nov, 1999, pp. 295-303.
- [24] 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, no. 9, pp. 574-586, September, 2004.
- [25] Zimmermann, T., Diehl, S., and Zeller, A., "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, June, 2005.