Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records

Mark Sherriff[1,2], Mike Lake[1], and Laurie Williams[2]
[1]IBM, [2]North Carolina State University
mark.sherriff@ncsu.edu, johnlake@us.ibm.com, williams@csc.ncsu.edu

Abstract

During development and testing, changes made to a system to repair a detected fault can often inject a new fault into the code base. These injected faults may not be in the same files that were just changed, since the effects of a change in the code base can have ramifications in other parts of the system. We propose a methodology for determining the effect of a change and then prioritizing regression test cases by gathering software change records and analyzing them through singular value decomposition. This methodology generates clusters of files that historically tend to change together. Combining these clusters with test case information yields a matrix that can be multiplied by a vector representing a new system modification to create a prioritized list of test cases. We performed a post hoc case study using this technique with three minor releases of a software product at IBM. 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.

1. Introduction

During development and testing, changes made to a system to repair a detected fault can often inject a new fault into the code base. These injected faults may or may not be in the same file(s) that were just changed, since the effects of a change in the code base can have ramifications in other parts of the system. New faults introduced during testing and maintenance can be isolated by impact analysis and regression testing techniques. Impact analysis and regression testing techniques exist that find changes in a system based upon a modification [12, 16]. However, techniques that utilize call graphs or other dynamic means can be computationally intensive, and may not be cost-effective for use in industry [18]. Often regression test selection techniques do not include files that are not part of the source code, such as properties files, help files, and configuration files. Additionally, current impact analysis and regression techniques are based upon semantic analysis and may not consider trends in actual usage and/or the fault-proneness of the set of files impacted [14, 15].

Insight into the injected faults due to fault removal techniques can 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 [9, 10]. Development artifacts can also provide information about underlying structures within a system that would normally not be apparent [2, 3, 7, 11, 24, 25].

Change record artifacts can show how files interact with one another or how a system is evolving during development by examining what areas of the system change together and where new sets of changes are emerging [2, 3]. Change records can be used to identify *association clusters* in a software system. An *association cluster* consists of a set of files that exhibit a specific relationship with the other files in the cluster with regards to particular development artifact [3]. If a number of files repeatedly change together to fix a set of faults, then those files would be identified as an association cluster. Association clusters are based upon historical patterns in the composition of revisions. We use singular value decomposition (SVD) [5] on a matrix of change record date to generate the association clusters.

The methodology proposed in this research is called SVD-Based Regression Test Prioritization (SRTP). SRTP provides a framework for gathering change records from testing and field failures, generating association clusters, and leveraging those clusters to guide regression test prioritization. Our goal is *to provide a methodology based upon change records and singular value decomposition that can prioritize regression tests to reduce the number of regression faults released to the field.* To examine the efficacy of our technique, we conducted an industrial case study with a project at IBM. Change records initiated from fault removal efforts were gathered on three consecutive minor releases of an IBM product. A *minor release* is defined as an "update and fix" pack to a final release of a product.

The rest of this paper is organized as follows. Section 2 provides information on background and related work. Section 3 describes the SRTP methodology in detail,

while Section 4 describes our case study at IBM. Section 5 presents a theoretical comparison with other techniques, and Section 6 gives a summary of this work.

2. Related Work

In this section, we will discuss related research and background literature in regression test selection, uses of clustering change records, and the singular value decomposition.

2.1 Regression test selection

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 [1]. The goal of regression test selection (RTS) techniques is to isolate the tests most likely to uncover injected faults after a system modification. The simplest RTS technique is the *retest-all* strategy, wherein the entire test suite is exercised. 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 [16]. 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 [22].

One safe RTS strategy was developed by Rothermel and Harrold, called `SelectTests` [16]. 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 unchanged 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 [18, 19].

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 [8].

Our technique adds to the body of knowledge of RTS 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.

2.2 Clustering files based upon change records

Research is currently being performed in gathering 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]. Ren et al. has created an Eclipse plug-in to predict the impact of code changes for developers to use in-process through white-box techniques [14]. Their plug-in, called Chianti, works by capturing atomic-level changes in the code base. Dependencies are then calculated between these atomic changes to predict what other areas of the code might be affected by a change through the use of call graphs. Ren performed two case studies on a 100 KLOC 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.

While Ren's works focused on guiding developer efforts, other research uses the same sets of change records to improve program comprehension. Further work expanded on the idea of gathering change records to isolate clusters of files within a software system to drive program comprehension. Beyer and Noack's work analyzed clusters of files using a co-change graph to plot files onto a two- or three-dimensional graph using an energy-based graph layout [3]. 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 [3]. Other research by Gall also generated association clusters of files within a system for program comprehension [7]. 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.

2.3 Singular value decomposition

Singular value decomposition (SVD) is a linear algebra technique that decomposes a given matrix into three component matrices [5]: (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 [5] 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*) [23]. 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* [23]. 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 [5, 23]. In our research, this core idea of isolating key components of the original matrix is the basis for using the SVD with SRTP. 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.

Osinski et al. created a clustering algorithm based upon SVD to improve search queries on a set of documents [13]. 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 [13]. 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 [13].

Using SVD to find association clusters amongst files differs from other forms of clustering in that the association clusters generated can overlap. Files may be related to different parts of the system in differing degrees of strength. Dickinson et al. investigated the use of various clustering techniques to create better sample sets from which to identify failures [6]. Research with several Java and C systems showed that certain techniques could cluster failures together and was more effective than random sampling for failures. Our technique is similar in purpose, in that the SVD will weigh more heavily those files that are failure-prone in the clustering process, thus making the detection of failures more likely.

3. SVD-based Regression Test Prioritization

SRTP 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 [20]. SVD is used to leverage its ability to illuminate underlying structures in a data set in which the data could be associated in multiple ways. In this section, we will describe the process of SRTP, which includes deriving the association clusters from change records and producing a reduced set of regression tests. Figure 1 outlines the steps of the SRTP method that will be further described in the remaining subsections.

```
1  Create matrix M where the values in the
matrix indicate the number of times two
files have changed together.
2  Create matrix T where the values in the
matrix indicate whether a file is affected
by a particular fault/test case.
3  [U, S, V] = svd(M);
4  for i:size of U
5    Gather cluster i information
6    for j:size of U
7       if |U(j, i)| > threshold
8         Place element of cluster i into R
9       end
10    end
11 end
12 Y = (R * R') * T;
13 Represent new system modification x as a
vector in which the value indicates whether
the file has been affected in this change.
14 Y * x yields the prioritization of each
test case based upon the similarity of the
new system modification x to previous
changes in the system.
```

Figure 1. Algorithm for SVD-based RTP.

3.1 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 are often linked together into revisions, 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 revisions, 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, revisions can be associated directly with the fault record that the changes are addressing, providing detail about how revisions are linked together. Information from fault tracking systems allows us to isolate revisions 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.

3.2 Gathering software development artifact data

After appropriate data sources have been identified, an analysis matrix can be generated that contains the systems files along each axis. The values within the analysis matrix 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 this analysis matrix as an example. Table 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 1. Sample Change Record Information.

Test Case ID	Fault ID	Revision 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 2. The values in the matrix represent the number of times that each file appeared in a revision with another file. Thus, 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 $\boldsymbol{M(2,2) = M(2,1) + M(2,3)}$). Similarly, File 3 has changed 21 times with File 2 and 3 times by itself.

$$M = \begin{array}{c} \\ F1 \\ F2 \\ F3 \\ F4 \\ F5 \end{array} \begin{array}{c} \begin{matrix} F1 & F2 & F3 & F4 & F5 \end{matrix} \\ \begin{bmatrix} 25 & 10 & 0 & 0 & 0 \\ 10 & 31 & 21 & 0 & 0 \\ 0 & 21 & 24 & 0 & 0 \\ 0 & 0 & 0 & 15 & 12 \\ 0 & 0 & 0 & 12 & 17 \end{bmatrix} \end{array} \quad \textbf{(2)}$$

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.

3.3 Perform the singular value decomposition

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 that the files changed together. A SVD of *M* provides the following matrices, shown in Equations 3 and 4:

$$U = V = \begin{bmatrix} -.29 & 0 & .9 & .31 & 0 \\ -.76 & 0 & -.02 & -.56 & 0 \\ -.59 & 0 & -.43 & .69 & 0 \\ 0 & -.68 & 0 & 0 & -.74 \\ 0 & -.74 & 0 & 0 & .68 \end{bmatrix} \quad \textbf{(3)}$$

$$S = \begin{bmatrix} 51.1 & 0 & 0 & 0 & 0 \\ 0 & 28.4 & 0 & 0 & 0 \\ 0 & 0 & 24.8 & 0 & 0 \\ 0 & 0 & 0 & 4.1 & 0 \\ 0 & 0 & 0 & 0 & 3.9 \end{bmatrix} \quad \textbf{(4)}$$

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 [23]. 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. In this example, 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.

3.4 Gather the association clusters

The values in the *U* matrix correspond to the composition of each association cluster. In this example, 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. 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. When a file's value is greater than the threshold, we can add that value to the matrix *R*, which will contain the final set of clusters. This is similar to the threshold that Osinski used in his algorithm [13]. 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. Matrix *R* from this *U* matrix is shown in Equation 5:

$$R = \begin{array}{c} \\ F1 \\ F2 \\ F3 \\ F4 \\ F5 \end{array}\begin{array}{c} \begin{array}{cccccccc} C1 & C2 & C3 & C4 & C5 & C6 & C7 & C8 \end{array} \\ \begin{bmatrix} .29 & 0 & .9 & 0 & .31 & 0 & 0 & 0 \\ .76 & 0 & 0 & .02 & 0 & .56 & 0 & 0 \\ .59 & 0 & 0 & .43 & .69 & 0 & 0 & 0 \\ 0 & .68 & 0 & 0 & 0 & 0 & .74 & 0 \\ 0 & .74 & 0 & 0 & 0 & 0 & 0 & .68 \end{bmatrix} \end{array} \quad \textbf{(5)}$$

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. Further, the third and fourth clusters collectively represent 22% (since these two clusters were split apart from the values in the third left singular vector) and the fifth and sixth represents 4%. These percentages show that the first cluster defines the majority of the information regarding these files. Clusters three, four, five, and six 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 *R* can provide information about the likelihood of a change in an association cluster based upon previous change information.

This technique is similar to the cluster rank algorithm used by Osinski et al. in their SVD-based search term clustering algorithm [13]. 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 [13].

3.5 Generate reduced test suite

The next step is to use the current revisions in the system to determine how each test case in the system is related to a given association cluster. Using data from the source control and fault management system, we can map a revision *x* directly to a particular fault in a one-to-one relationship. To accomplish this analysis, we can take the set of all revisions in the source control system and create a matrix *T*, with each row indicating a revision and each column representing a particular file. An example matrix *T* has been constructed from the information in Table 1 in Equation 6.

$$T = \begin{array}{c} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \\ T6 \end{array}\begin{array}{c} \begin{array}{ccccc} F1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \quad \textbf{(6)}$$

Since a revision is opened by a single distinct fault, we can say that each row represents a specific fault found in the system. 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, revisions 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 7:

$$T * R = [Tests\ x\ Files] * [Files\ x\ Clusters] = [Tests\ x\ Clusters] = P \quad \textbf{(7)}$$

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 revision.

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 8:

$$R * P^T = [Files\ x\ Clusters] * [Clusters\ x\ Tests] = [Files\ x\ Tests] = Y \quad \textbf{(8)}$$

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 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. In this multiplication, each the value of how relevant a test case is to a file (with 0 being no relevance and 1 being a one-to-one link between the two) is added together based upon which files are in the new revision. An example of this multiplication is shown in Equation 9:

$$Yx' = \begin{array}{c} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \\ T6 \end{array}\begin{array}{c} \begin{array}{ccccc} F1 & F2 & F3 & F4 & F5 \end{array} \\ \begin{bmatrix} .69 & .09 & .23 & 0 & 0 \\ .32 & .90 & .76 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ .78 & .60 & .62 & 0 & 0 \\ .32 & .90 & .76 & 0 & 0 \end{bmatrix} \end{array} * \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} .32 & 1.7 & 0 & 0 & 1.2 & 1.7 \end{bmatrix} \quad \textbf{(9)}$$

The value in the result priority 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. This effect occurs because of the way that the SVD takes into account how all files tend to change together. 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. For example, if more files have changed, the priority values can be higher since there is now an additional component to match with available tests. 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.

Faults 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 faults 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 revisions, will have a greater impact on the values in the result vector.

3.6 Limitations

The association clusters are based upon the change records that may or may not be accurate. For example, the case study reported in this paper 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 whereby a developer makes changes unrelated to an open fault while fixing that fault. If opportunistic changes occur during fault removal efforts, we cannot be certain that the set of files that change together are related to a particular fault. However, we have noticed in ongoing work that developers that use continuous code integration and make smaller changes to the code base tend to avoid having opportunistic changes.

Another limitation is that our technique requires that some level of 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. In our study, this data exists as an additional note in the description of the fault in the fault tracking system.

4. IBM Case Study

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 revisions and both pre- and post-release failures.

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 revision is to a given test case, but also the impact that a test could have on surrounding files in an association cluster. 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 revisions for each release as the "historical data" from which we generated the association clusters. From the remaining one-third of the revisions, twenty revisions were selected for use as "future changes" that are being introduced into the system and for which we are prioritizing regression tests. The remaining revisions 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 count this as a confirmed true positive result.

For each of the twenty revisions selected to represent new changes introduced into the system, we converted these revisions into vectors. Each of these vectors was then multiplied by the association matrix *Y* generated from the "historical data" set of revisions. 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" revisions with the future set of faults to determine if the 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 2.

We found that 50% of the time our technique did not recommend any additional tests beyond a simple retest of changed files. If the model recommends the same tests as a simple retest, this indicates that the areas of the system that these revisions 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. 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 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.

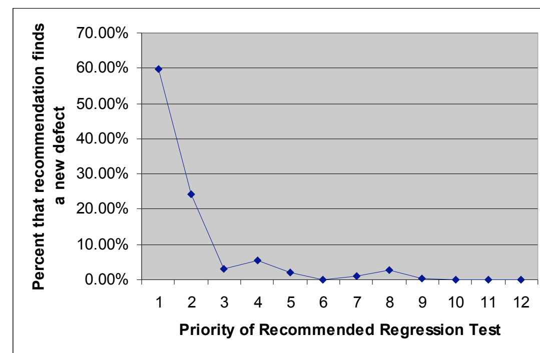

Figure 2. Percentage matches in future data by regression test priority.

5. Comparison to Other Techniques

In this section, we 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 [15]. 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[*];

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.

5.1 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 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 [15].

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 [15, 17]. 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. When the an image file 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 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.

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

5.2 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 [15]. Due to the use of historical change information, 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 [18].

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 [15]. 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.

5.3 Efficiency

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 $O(R*A)$, where R is the number of revisions and A is the average number of files in a revision. 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 $O(F \log F)$, where F is the number of files in the system. 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 $O(F*C)$, where C is the number of files in that revision.

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. 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.

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.

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 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. 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.

6. Summary

In this paper, we explored an empirically-based regression test prioritization method based upon structures discovered through change records and singular value decomposition. To show the efficacy of our technique, a case study was performed with three releases of a product from IBM. 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. 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.

7. Acknowledgments

We would sincerely like to thank J.B. Baker at IBM for his input into this work. Partial funding was provided for NCSU authors by the National Science Foundation.

8. References

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

[2] Ball, T., Kim, J., Potter, A., and Siy, H., "If your version control system could talk," in *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," in *13th IEEE International Workshop on Program Comprehension*, St. Louis, MO, 2005, pp. 259-268.

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

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

[6] Dickinson, W., Leon, D., and Podgurski, A., "Finding Failures by Cluster Analysis of Execution Profiles," in *International Conference on Software Engineering*, Toronto, Canada, 2001, pp. 339-348.

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

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

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

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

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

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

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

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

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

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

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

[18] 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, June 1998.

[19] 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, October 2001.

[20] 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.

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

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

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

[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, 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, June 2005.