Towards a More Efficient Static Software Change Impact Analysis Method

Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
{a.jashki,r.zafarani,ebagheri}@unb.ca

ABSTRACT

Impact analysis methods are commonly employed to reduce the likelihood of encountering faulty or unexpected behavior from a software program as a result of developers' oblivious modifications. In this paper, we propose a static impact analysis technique that creates clusters of closely associated software program files based on their co-modification history in the software repository. The proposed method benefits from dimensionality reduction techniques to reduce the complexity of the collected information and perform the impact analysis process faster. The method has been tested on four different open source project repositories, namely Firefox, Firebird, Thunderbird, and FileZilla. The results of the impact analysis method performance in terms of precision (impact set identification accuracy) and execution time cost have been reported in this paper. The proposed method shows promising behavior when used with several specific clustering techniques such as DBscan and X-Means.

1. INTRODUCTION

During its lifecycle, a software program may experience various releases through which new features are added, bugs are fixed or new requirements are satisfied. In order to accommodate such changes to the software, not only do developers add new modules to the program, but also alter the internals of the existing code to be able to suit the new conditions. Subsequently, as the program gets larger, more complicated, and involves multiple developers, the potential impact of change may not be fully understood by a developer making a certain modification on the software program; therefore, impact analysis techniques are often used to determine the potential effect of a proposed software change on a subset or entirety of a given software program [2].

Impact analysis techniques can be performed before a modification has been actually implemented to estimate the probable costs of the proposed change, which allows for a predictive cost-benefit analysis. Furthermore, such techniques can also be employed after software revision in order to identify the potential software modules requiring retesting and/or update [17]. In this way, impact analysis methods reduce the likelihood of encountering faulty or unexpected behavior from a software program as a result of oblivious modifications.

Many impact analysis techniques have been proposed in the related literature [1, 4, 12] that mainly view impact analysis as a process of extracting module/code interdependencies through mining call graphs or execution paths. Such techniques have exhibited reasonable performance with regards to their precision while also being effective in reducing the number of false-positives; hence, such techniques will avoid the over-estimation of the effect of a software change. Despite their useful properties, these techniques fall short in satisfying the requirements of impact analysis techniques under the following two conditions:

- As we will show later in this paper, most software programs consist of a wealth of non-source code files including media files, help files, configuration files and others. Although these files are important for the proper operation of the program, they do not appear in call graphs or program execution paths; and therefore, go unnoticed in such impact analysis techniques. As an example, suppose that the database access layer of a software program has been modified to accommodate the migration to a new DBMS, but the related configuration files have not been updated to reflect this change. This will result in a failure whose cause is not easy to spot.
- Many software applications are recently developed for web-based platforms using declarative XML-based languages as opposed to procedural languages. For instance, the user interface of many programs are defined using HTML, CSS or XUL, among others. In these languages, function-calls are not directly used, instead construct inclusion axioms are often employed. Due to the nature of this programming paradigm, program execution paths may be insufficient for identifying the potentially impacted files of such programs. This is expected since the debugging of web-based applications is quite different from programs written in normal procedural languages.

To address these issues, we take an approach similar to the method introduced by Sherriff et al. [20]. In our proposed approach, we mine software change repositories (e.g. the CVS) to find clusters of closely related files without considering the program execution paths. The basic assumption

PASTE'08, Atlanta, Georgia USA.

behind our approach is that *those files that are committed to the software repository in a single working session by the same developer possess some form of conceptual similarity and interdependency.* Simply stated, since developers often focus on modifying a single functionality/aspect of the software entity at a given point of time, and they tend to commit the performed changes once they are finished with an important step in the modification, the files in the same commit session can be considered to be related. Based on this assumption, we will consider those files that have been frequently seen together in the commit sessions to have impact on each other; therefore, in the impact analysis process, given some initial file such as f, our approach will propose a set of files such as $\mathcal{F}$ that have been previously frequently seen with f in the commit history.

The technique we propose in this paper provides a methodology for creating clusters of associated files within any given software repository. The method examines the history of file changes and creates a matrix, which stores the frequency of mutual file manipulations derived from the change logs. For a software repository consisting of n files, an $n \times n$ matrix is created, representing the two-by-two frequency of simultaneous file changes. An interesting observation from many software repositories shows that the developed matrix is usually *sparse*, which means there are only a limited number of files that are frequently seen together that will eventually form the clusters. Since the frequency matrix is sparse, we can easily reduce its dimensionality and form a new matrix such as $\hat{M}$ of size $n \times \hat{n}$, where $\hat{n}$ denotes the reduced dimension size. We then employ various clustering methods to form the associated file clusters from $\hat{M}$.

This paper provides three contributions: 1) a new metric for measuring the conceptual distance of two files from software change logs is given; 2) dimensionality reduction techniques are introduced to reduce the execution time of the impact analysis process; and 3) a comparative analysis of the performance of various clustering techniques for static impact analysis is provided. The paper is structured as follows: The next section will review some of the important background literature on software impact analysis. Section 3 describes the details of the proposed method. The experimental evaluation of the method on four different open source software projects is given in Section 4 and the paper is concluded in Section 5.

2. RELATED WORK

Impact analysis techniques can be broadly categorized into two classes, namely *static* or *dynamic*. Dynamic impact analysis methods consider information collected from the runtime execution of the software in order to draw relevant conclusions. Such information can be gathered by brute-force execution of the program under all possible circumstances or by test suites with an instrumented code base [15, 16]. On the other hand, static impact analysis techniques do not consider information from program execution, instead they function over data acquired through the software development lifecycle or the semantic analysis of the program source code [24, 19].

`CoverageImpact` [15] and `PathImpact` [16] are two important dynamic impact analysis methods. The `CoverageImpact` method depends on lightweight instrumentation to gather runtime software execution information. The instrumented program collects program coverage information in the form of bit vectors (one bit per method). Various bit vectors are accumulated for different execution traces, which are used in three consecutive steps to derive the impact set: 1) A *covered set* is created that includes all the methods within those execution traces that include a given method such as m; 2) A *slice set* is formed that incorporates all the methods within a static forward slice from m; 3) Finally, an intersection of the *slice set* and *covered set* is calculated, which represents the required *impact set*. Similarly, the `PathImpact` method also employs instrumentation to collect information from a running software program. Subsequently, this method employs the SEQUITUR compression algorithm to derive a condensed representation of the execution traces called a whole-path directed acyclic graph (DAG). `PathImpact` traverses the whole-path DAG instead of the execution traces to find the *impact set* given a set of changes. The method starts at some changed method in the graph and performs recursive forward and backward in-order traversals at each node of the whole-path DAG and stops whenever an execution trace termination symbol is encountered.

Orso et al. have shown that although these two algorithms perform similarly, `PathImpact` is slightly more precise than `CoverageImpact`, but is more expensive in terms of time usage and space consumption [17]. `CoverageImpact` and `pathImpact` have two minor deficiencies. First, since a lot of software failures rise as a result of *low probability-high impact* conditions within the code and test suites may not inclusively capture all these conditions, such methods can be less than perfect for identifying critical points of failure under the mentioned circumstances; therefore, the success of these methods is dependent on the comprehensiveness of the test suites. Second, these two methods do not consider the weight of the methods frequently observed in various execution traces, i.e., if two methods are only encountered once in some execution trace, they will be given equal importance as two other methods that are seen much more frequently in many execution traces; hence, it is difficult to accurately determine the precedence of the members of the *impact set* in such impact analysis methods.

Within the static impact analysis paradigm, Zimmerman et al. [24] have developed a real-time impact identification eclipse plug-in that enables the developers to quickly see the ripple effects of the modifications they are currently making. The plug-in mines source code repositories in order to create transaction rules, which indicate areas of the software that tend to change together. Their plug-in is able to correctly predict 44% of the related files to a given change on a stable software code base. Similarly, Ren et al. have also created an Eclipse plug-in to predict the impact of code level changes on other areas of the software program. The plug-in, called Chianti, captures atomic-level code modifications and creates call graphs. Call graphs are then employed to predict other areas of the software that may be impacted. Experimental results showed that Chianti was able to reduce the number of regression test depending on the degree of performed modifications [19].

The work presented by Sherriff et al. is very similar to our proposed work [20]. The authors employ Singular Value Decomposition (SVD) in order to form association clusters of related files within a software program. We build on their initial assumptions and show in this paper that methods based on SVD are more time-consuming and less accurate than other important clustering methods. Within the same

line of work, Kagdi et al. [13, 14] have proposed the use of sequential-pattern mining algorithms that take a set of sequences and find all the frequently occurring subsequences that have at least a user-specified minimum support. These pattern mining techniques are applied on to the commits in software repositories for uncovering highly frequent co-changing sets of artifacts to reveal traceability link between the files. They base their work on a similar assumption to our work that 'if different types of files are committed together with high frequency then there is a high probability that they have a traceability link between them'. Anlagous to this work, Ying et al. [23] have proposed a three staged approach to determine change patterns. In their approach, they initially extract the data from a software configuration management system and preprocess the data to make it suitable for a data mining algorithm. In the second stage, an association rule mining algorithm is employed to form the change patterns, and finally, relevant source files as part of a modification task by querying against mined change patterns is recommended. As it can be seen, these work tackle the problem addressed in this paper from a different perspective. They attempt to formalize the process of finding program change patterns through the employment of techniques in the field of association rule mining. In contrast, we devise a closeness metric in this paper, which would serve as a distance measure in the process of clustering closely related files. We will introduce our method in the next section. Some other useful methods that mine software change records to increase program comprehension can also be found in the related literature [8, 9, 7].

3. THE PROPOSED METHOD

The static impact analysis method introduced in this paper attempts to find clusters of closely associated files of a software program through four main steps:

1. The software change repository (CVS) is mined for frequently co-occurring files. A matrix is developed that contains the degree of closeness of any two files within the software program.

2. An intrinsic dimensionality estimation method is performed on the developed matrix to estimate a lower dimensional representation of the it.

3. Principle Component Analysis (PCA) is executed on the matrix based on the estimated lower dimension, and consequently a much smaller matrix is created.

4. The rows of the compressed matrix are employed as the coordinates of each file in the vector-space. The distance of each two file is calculated and passed onto some clustering algorithm in order to identify the clusters of associated files from the modules of the software program.

We will discuss the details of each step in the following subsections.

3.1 Co-occurrence Matrix

In order to find closely related files within the scope of a software program, we need to observe the number of times that each set of files have been observed together in the history of modifications of the software. For example, if $File_2$ is frequently changed immediately after a change in $File_1$, we may be able to infer that these two files are either syntactically or conceptually related. The more these two files are seen together, the higher our confidence in their inter-relationship would be. For this reason, Sherriff et al. have proposed to count the number of times that two given files have been seen together as an indicator of their closeness [20]. Based on this assumption, a matrix such as $M_{n\times n}$ can be built in which n denotes the number of files of the software program; therefore:

$$M[i,j] = \mathcal{N}(F_i \cap F_j). \tag{1}$$

where $\mathcal{N}(F_i \cap F_j)$ represents the number of times that file F_i has been observed with file F_j in the same session. This definition of closeness of two files cannot distinguish between the frequency of co-occurrence of two files and the number of times that two files have been previously observed together. In other words, lets suppose $\mathcal{N}(F_1 \cap F_2) = 5$, and $\mathcal{N}(F_1 \cap F_3) = 9$. With the above definition, F_1 is closer to F_3 than to F_2. But further assume we have the following information: $\mathcal{N}(F_1) = 10$, $\mathcal{N}(F_2) = 5$, and $\mathcal{N}(F_3) = 100$, which means that F_1, F_2, and F_3 have been seen a total of 10, 5, and 100 times, respectively. With this new information, it is clear that F_1 and F_2 are closer to each other than to F_3, since F_2 has always appeared with F_1 in all of its occurrences in the history of changes.

To accommodate such cases, we alter the definition for $M[i,j]$ as follows:

$$M[i,j] = \frac{\mathcal{N}(F_i \cap F_j)}{\mathcal{N}(F_i) + \mathcal{N}(F_j) - \mathcal{N}(F_i \cap F_j)}. \tag{2}$$

The modified version of the closeness metric is now able to differentiate between the *frequency of co-occurrence of two files* and the *number of times they have been observed together*.

3.2 Intrinsic Dimensionality Estimation

Lets suppose a software program consisting of $20,000$ files is being analyzed. The size of its co-occurrence matrix would be $20,000 \times 20,000$, which is too large for most algorithms to process. In light of the fact that the co-occurrence matrix of most software programs is sparse, which means that many of the files of the program have not been edited simultaneously, we can reduce the dimensions of this matrix in order to make its manipulation more convenient. For this purpose, we need to estimate the most suitable lower dimension for the given matrix such that its current properties are best preserved. Most of the techniques for estimating the intrinsic dimensionality of a given dataset are based on the intuition that data lies on or near a complex low-dimensional manifold that is embedded in the high-dimensional space; hence, the techniques for dimensionality estimation aim at identifying and extracting the manifold from the high-dimensional space [22].

In order to estimate the intrinsic dimensionality of the co-occurrence matrix, we employ the Eigenvalue-based estimator [22]. The eigenvalue-based intrinsic dimensionality estimator performs principle component analysis on the high-dimensional matrix and evaluates the eigenvalues corresponding to its principal components. The estimation of the intrinsic dimensionality of the co-occurrence matrix is performed by counting the number of normalized eigenvalues that are higher than a given small threshold value. Given the

co-occurrence matrix, the estimator will provide the most suitable lower dimension value ($\hat{n}$) for the matrix.

3.3 Dimensionality Reduction

Based on the recommendation received from the Eigenvalue-based estimator, the co-occurrence matrix $M_{n\times n}$ needs to be converted into a new representative matrix $\hat{M}_{n\times\hat{n}}$. In order to perform the transformation from $M \rightarrow \hat{M}$, we employ the Principle Component Analysis (PCA) method to move to the new dimension ($\hat{n}$) provided by the estimator. PCA is a method that can be used for dimensionality reduction in a matrix by preserving those features of the data that contribute most to its variance, by keeping lower-order principal components and ignoring higher-order ones. The low-order components often contain the most important aspects of the input data [5].

As a result, the following transformation is performed on the co-occurrence matrix:

$$\hat{M}_{n\times\hat{n}} = PCA(M_{n\times n}, \hat{n}). \quad (3)$$

3.4 File Representation and Clustering

The principle component analysis of the co-occurrence matrix provides a lower dimensional representation of it that can be more easily manipulated. The files now need to be mapped to a vector-space model through which the coordinates of each file can be used to determine its location in the multidimensional space. For this purpose, the $\hat{n}$ values in the columns of the matrix related to each row are taken as the $\hat{n}$-dimensional representation of each file; hence, the coordinates of any file such as $File_i$ in the vector-space model is as follows:

$$\overrightarrow{File_i} = (\hat{M}[i,1], \hat{M}[i,2], ..., \hat{M}[i,\hat{n}]). \quad (4)$$

It is now possible to determine the closeness of any two given files of the software program based on the distance of their corresponding locations in the $\hat{n}$-dimensional space. To calculate the distance of two files from each other, we employ the Cosine similarity measure as follows:

$$\mathcal{D}(File_i, File_j) = \frac{File_i.File_j}{|File_i|.|File_j|}. \quad (5)$$

where $\mathcal{D}(File_i, File_j)$ denotes the distance between $File_i$ and $File_j$. Being able to determine the degree of closeness of the files in terms of their distance in the vector-space model, we can now cluster the files of the software program in order to detect the groups of closely associated files within the software. The members of the same clusters would be considered as closely associated and hence become the members of the same *impact set*. In this paper, we employ the following five clustering techniques to partition the program files into relevant impact sets:

K-means The K-means algorithm assigns each point to the cluster whose centroid is nearest to that given point. The centroid is the average of all the points within that cluster, i.e, its coordinates are the mean of each dimension separately over all the points in the cluster; therefore, the centroid of each cluster dynamically changes until it converges to a stable point. K-means requires as input the optimal number of clusters to be able to perform properly [10].

X-means This algorithm efficiently searches the space of cluster locations and the number of clusters in order to optimize the Bayesian Information Criterion (BIC) or the Akaike Information Criterion (AIC) measure. X-means is much faster than repeatedly using accelerated K-means for different values of K [18].

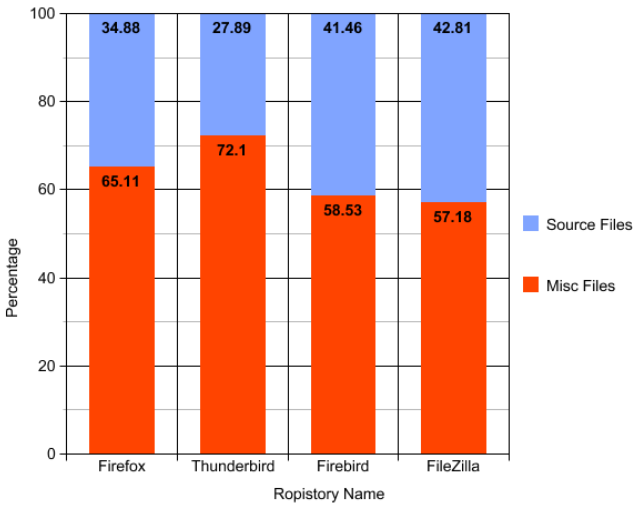

Figure 1: The ratio of source and non-source files in each object of analysis.

Spectral Clustering The Spectral clustering techniques make use of the spectrum of the similarity matrix of the data to perform dimensionality reduction for clustering in fewer dimensions. We employ the Shi-Malik algorithm in this paper [21].

EM The EM clustering algorithm computes probabilities of cluster memberships based on several probability distributions. The goal of this clustering algorithm is to maximize the overall likelihood of the data, given the final clusters [3].

DBscan DBscan is a density based spatial clustering method that clusters data of any arbitrary shape. DBscan is based on two main concepts: density reachability and density connectability. These two concepts depend on two input parameters of the DBscan clustering algorithm: the size of epsilon neighborhood and the minimum points in any cluster [6].

The clustering methods will group the software program files into several clusters, which we will consider to be the *impact sets*.

4. EXPERIMENTAL EVALUATION

In this section, we will initially introduce the software datasets that were employed within our experiments and then will describe the experimental setup and the obtained results.

4.1 Objects of Analysis

As objects of analysis, we used four different widely used open source projects, namely Firefox, Thunderbird, FileZilla and Firebird. Firefox is a web browser managed by the Mozilla Corporation. Firefox had gained 19.03% of the usage

Table 1: Objects of Analysis.

Project Name	# of Files	# of Commits	Avg # of Revisions
Thunderbird	1871	14171	7.574
FileZilla	967	8122	8.39
Firefox	2950	31003	10.5
Firebird	1336	3323	2.48

share of web browsers as of June 2008, making it the second-most popular browser in current use worldwide. Thunderbird is a cross-platform e-mail and news client developed by Mozilla. As of June 2008, Thunderbird has been downloaded more than 67 million times since 1.0 release. FileZilla is a cross-platform FTP client. It supports FTP, SFTP, and FTPS (FTP over SSL/TLS). As of June 2008, it was the 10^{th} most popular download of all time from SourceForge.net. Firebird is a relational database management system offering many ANSI SQL:2003 features[1]. The break-up of the details of each of these projects in terms of the number of project files, number of commits to the repository, and the average number of file revisions can be seen in Table 1.

Figure 1 reveals that in all four projects the number of non-source files is more than the number of source files, which shows that in such software projects static impact analysis methods are more capable of identifying the relationship between the non-source files of the program since such files do not appear in execution paths or call graph for dynamic impact analysis methods to be able to process them.

4.2 Experiment Design

In order to gather the frequency of co-occurrence of the files within each of the four projects, we analyzed the CVS repository of these projects. The versioning system maintains a comprehensive history file that contains all the details of modifications and operations performed on any given file. For this reason, we parsed the history file and extracted all of the revision records of the project files. Each record of the history file related to a given file contains the timestamp, the name of the person modifying the file, the file version, and other relevant information. Based on these information, we reconstructed only the commit sessions of the version control system such that all the files committed to the repository by the same person in the exactly the same time were considered to be in the same *session*.

The number of times that a given file such as f_i was encountered in the history file was used as $\mathcal{N}(f_i)$, and the number of times that two file (f_i and f_j) had been observed in the same session was considered to be $\mathcal{N}(f_i \cap f_j)$. These information were employed to create the co-occurrence matrix, which was subsequently given to the Eigenvalue-based dimensionality estimator. The matrix was then reduced in dimensionality by PCA and passed onto various clustering algorithms to create the impact sets.

The proposed static impact analysis technique is written in Java. It includes run-time communication with Matlab to perform dimensionality estimation and reduction. The Matlab Toolbox for Dimensionality Reduction developed by van der Maaten [22] was employed to perform the Eigenvalue-based dimensionality estimation and principle component

[1]Description of these projects are from wikipedia.com

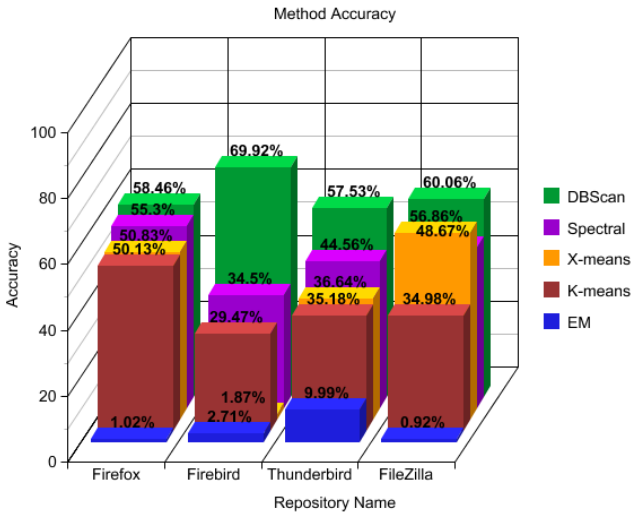

Figure 2: Precision results for the different objects of analysis.

analysis. The obtained results were then processed in Java using several clustering algorithms developed in the Weka framework [11]. All the results related to the running time and precision of the methods were obtained from a program running on a dedicated 2.8 GHz Pentium 4 Personal Computer with 2 GB of memory, running a Windows XP operating system along with Java 6.0 and Matlab 8.0.

4.3 Results and Analysis

In the following two subsections we will analyze the performance of each of the variations of the static impact analysis method using different clustering algorithms in terms of their precision and execution time.

4.3.1 Precision

Figure 2 summarizes the precision results obtained for the different software programs using the various combinations of the impact analysis method. The precision of the methods were calculated based on a 10 fold cross validation scheme[2] where the CVS history file was divided into two sections: train history section and the test history section. The train section of the history file was employed to create the co-occurrence matrix and develop the cluster model of the data. Once the clusters were created the test section was employed to calculate the precision of the developed impact sets. For this step, a file (f_i) was taken from the test section and its corresponding location within the developed cluster model was determined. Once the proper estimated impact set (cluster number where the file is placed) was identified, the files in the same session with f_i in the test set were gathered. Since these files appear with f_i in the test set, we expect our learnt model to be able to predict these files in the impact set. Therefore, we look for these files within the estimated impact set. The precision of the method is therefore calculated as the ratio of the number of correctly predicted files by the learnt model over the total number of files in the same session with f_i in the test set. We perform

[2]http://en.wikipedia.org/wiki/Cross-validation

Table 2: Execution time in milliseconds.

Project Name	EM	X-Means	Spectral Clustering	K-Means	DBscan
Thunderbird	121282	63	749937	62	1656
FileZilla	81141	31	20891	32	172
Firebird	609250	78	570031	63	1172
Firefox	96015	94	3684078	125	4079

this process for all the files in the test set. The final precision of the method is the average precision obtained for all the files in the test set.

Lets suppose f_s^i is the i^{th} file of session s of the test section. Further assume that session s contains n_s files and the test section includes p files in total. If f_s^i is located in the k^{th} cluster of the learnt model, then precision of the learnt model for f_s^i is defined as follows:

$$precision(f_s^i) = \frac{\sum_{l=1}^{n_s} \Phi(f_s^l, k)}{n_s}. \quad (6)$$

where $\Phi(f_s^l, k)$ is a boolean function (returns 0/1) that checks to see whether a given file such as f_s^l is a member of the k^{th} cluster of the learnt model or not. In other words, it checks to see if the files that are seen in the same session with f_s^i in the test section of the history, do also appear in the learnt impact set. Now, the overall precision can be calculated:

$$precision = \frac{\sum_{i=1}^{p} precision(f_s^i)}{p}. \quad (7)$$

Here, precision is computed as an average over all single precision values of each session.

The precision results are shown in Figure 2. The obtained precision results are promising (specially the results of DBscan), since they outperform the results reported in [17, 20]; however, these results are not directly comparable since the objects of analysis are different.

4.3.2 Execution Time Cost

In addition to a reasonable accuracy, the static analysis method must also benefit from a suitable execution time for it to be useful in realtime application domains. For instance, suppose that the proposed impact analysis method is to be incorporated into an Eclipse plug-in, in such a case it needs to be fast enough to be usable by software developers. Table 2 reports the execution time of the different variations of the proposed impact analysis method on the objects of analysis. It should be noted that the time to perform dimensionality reduction has not been included in these figures, since the dimensionality reduction time is a unique constant time similar for all of the methods. The execution time has been recorded based on the time taken for the clustering algorithm to create the impact sets, followed by a find operation to seek the appropriate impact set for a given file. All data reported in Table 2 are in milliseconds.

As it can be seen the variation of the proposed method that employs the X-means[3] clustering algorithm is the fastest, and the Shi-Malik Spectral clustering method [21] is the slowest. Generally, clustering methods that use some form of singular value decomposition such as Spectral clustering methods perform very slowly due to their high time complexity; therefore, the method proposed by Sherriff et al. that employs singular value decomposition for forming the impact sets may be too time-consuming for real-time application domains.

[3]Since k-means requires the number of optimal clusters as input, its actual execution time is 20 times slower than the value reported in Table 2, since we initially ran K-Means for 20 different cluster sizes in order to find the best cluster size.

4.3.3 Trade-off Analysis

Precision and execution time are two important factors in selecting the most appropriate variation of the proposed method. A desirable method should be able to provide reasonable results within a tolerable execution time/delay. Consider the case (similar to Chianti) that the developers are employing an Eclipse plug-in to identify the impacted areas of a software program by some modification. In these cases, the developers need to see the information in a real-time fashion; therefore, methods similar to Spectral clustering, which takes close to one hour to provide a feedback, are not viable solutions.

DBscan, K-Means, and X-Means have shown to both have good precision and acceptable execution time. Overall, DBscan has performed best on all of the four software programs and also provides reports in a very short amount of time; therefore, it can be considered a good candidate for performing static impact analysis.

5. CONCLUDING REMARKS

In this paper, we have presented a method for the static impact analysis of software programs. The proposed method incorporates four novel features: 1) a co-occurrence measuring metric that is able to distinguish between the frequency of co-occurrence of two files and the number of times the two files have been observed together in the software modification history; 2) the employment of the Eigenvalue based dimensionality estimator and the principle component analysis technique to reduce the size of the co-occurrence matrix for faster impact analysis; 3) a vector-space representation of the software program files in a multidimensional space and the use of the Cosine similarity metric to find the degree of closeness of two files; and 4) the exploitation of five different clustering methods to create the required impact sets.

As future work, we are interested in empirically comparing our co-occurrence metric with the metric proposed in [20]. Also, we are interested in evaluating the performance of our proposed method in comparison with several dynamic impact analysis methods such as PathImpact and CoverageImpact. It would also be interesting to combine our proposed approach with several dynamic impact analysis methods. This way the our static impact analysis technique would reduce the number of source files that need to be searched by the dynamic models. Finally, we believe that it will be beneficial to reduce the dimensionality of the created impact sets to 2, in order to be able to visualize them in 2D space.

6. REFERENCES

[1] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 432–441, New York, NY, USA, 2005. ACM.

[2] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[3] P.S. Bradley, U. Fayyad, and C. Reina. Scaling EM (Expectation-Maximization) Clustering to Large Databases. *Microsoft Research Report, MSR-TR-98-35, Aug*, 1998.

[4] Ben Breech, Mike Tegtmeyer, and Lori Pollock. A comparison of online and dynamic impact analysis algorithms. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Songcan Chen and Yulian Zhu. Subpattern-based principle component analysis. *Pattern Recognition*, 37(5):1081–1083, 2004.

[6] M. Ester, H.P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press*, pages 226–231, 1996.

[7] Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 66–74, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[9] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[10] JA Hartigan and MA Wong. A K-means clustering algorithm. *JR Stat. Soc., Ser. C*, 28:100–108, 1979.

[11] G. Holmes, A. Donkin, and I.H. Witten. Weka: a machine learning workbench. *Intelligent Information Systems,1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361, Nov-2 Dec 1994.

[12] Lulu Huang and Yeong-Tae Song. Dynamic impact analysis using execution profile tracing. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 237–244, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Huzefa Kagdi, Jonathan I. Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 145–154, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, New York, NY, USA, 2006. ACM.

[15] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 430, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137, New York, NY, USA, 2003. ACM.

[17] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.

[18] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, 1929.

[19] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 664–665, New York, NY, USA, 2005. ACM.

[20] Mark Sherriff and Laurie Williams. Empirical software change impact analysis using singular value decomposition. *First International Conference on Software Testing, Verification, and Validation*, 0:268–277, 2008.

[21] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, Aug 2000.

[22] L. J. P. van der Maaten, E. O. Postma, and H. J. van den Herik. Dimensionality reduction: A comparative review. 2007.

[23] Annie T. T. Ying, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004. Member-Gail C. Murphy.

[24] Thomas Zimmermann, Member-Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.