Studying the Evolution of Software Systems Using Change Clusters

Jay Kothari, Trip Denton, Ali Shokoufandeh, Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{jhk39, tdenton, ashokouf, spiros}@cs.drexel.edu

Ahmed E. Hassan
Performance Engineering
Blackberry Enterprise Software
Research In Motion (RIM)
Waterloo, Canada
ahmed@myblackberry.com

Abstract

*In this paper, we present an approach that examines the evolution of code stored in source control repositories. The technique identifies **Change Clusters**, which can help managers to classify different code change activities as either a software maintenance or a new development. Furthermore, identifying the variations in **Change Clusters** over time exposes trends in the development of a software system.*

*We present a case study that uses a sequence of **Change Clusters** to track the evolution of the PostgreSQL software project. Our case study demonstrates that our technique reveals interesting patterns about the progress of code development within each release of PostgreSQL. We show that the increase in the number of clusters not only identifies the areas where development has occurred, but also reflects the amount of structural change in code. We also compare how the **Change Clusters** vary over time in order to make generalizations about the focus of development.*

1 Introduction

Managers responsible for large software systems are always in search of techniques to measure and quantify the development trends in a project. For example, the complexity of the source code or the number of "bad smells" [8] over time are used to estimate the health of the source code and the need to schedule future refactoring activities. Similarly, the number of reported bugs and applied fixes are often used to determine the readiness of a software system for release. These techniques help to ensure the long-term health of the software system and reduce the cost of its maintenance.

We are interested in techniques that give managers an overview of the code development process enabling them to characterize the major work activities during a time period. Finding trends in these work activities will allow managers to better understand the software system's life-cycle and help them plan their development activities accordingly. We seek to describe source code changes automatically. While it is possible to retrieve atomic changes in the code from source control repositories, this information is overwhelming and requires in depth knowledge of the system to comprehend. Even though terms such as perfective, corrective, and adaptive are used to describe changes to the code; it is not possible to describe changes to a software system using these terms in a confident, accurate, and automated fashion. It will require numerous heuristics, human intervention, and intuition to rank changes to source code accordingly. In short, we seek an approach that provides a balance between the expressiveness of the recovered descriptions and the ease of automating the recovery process.

Consider a progress report from a development team. Rather than provide specific details regarding the individual changes to the code, or a biased interpretation of the activities on the software system we are interested in objective measurements. These measurements should provide not only the areas of work, but a statistical description of the amount of work in those areas. Furthermore, they should be consistent regardless of the individual reporting it and should present no bias based on an intimate knowledge of the product's development. For example, a manager of a team working may provide the following measures of progress:

1. We changed over 700 lines of code.
2. We added 400 lines of code and removed 300 lines of dead code.
3. We modified code in four subsystems.

The first reply deals with changes to the overall size of system. The second reply specifies the addition and deletion of code. The third reply is even more specific than the second reply as it maps the changes to the exact subsystems. Instead, a more informed manager interested in providing

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE

a more informative description of the progress of his team might say:

1. All the development that has occurred can be categorized into four distinct areas. Furthermore, for each of these areas, there is one change that best represents all the changes that have been applied to the system in that particular area.

2. One of the areas where development occurred is software maintenance since we have seen areas very similar to it previously. However, since we have not previously seen areas similar to the other three areas we found, they are clearly new development.

3. By associating every change that was applied to the system during this phase we can find a distribution of change activities and by proxy, efforts. We can also determine how much overall maintenance occurred versus new development.

This reply groups the changes under four different change categories, classifies all the changes under those categories, and presents the distributions of all the changes that were applied to the system.

Our approach examines the temporal evolution of code stored in source control repositories. It employs the notion of "canonical sets" to identify a subset of *Canonical Changes* that best represent the modification activities within a time period. These *Canonical Changes* act as central points for all of the modifications applied to a system in a given time period, inducing a clustering of all of the modifications. We call the created clusters – *Change Clusters*.

Using this clustering we can discover the distribution of effort across the various change categories. For example, even though our canonical set analysis may identify 10 different change clusters during a time period, it may be the case that most changes belong to one or two clusters and the other clusters contain very few changes. By studying the distribution of effort, managers can discover if their team is spread thin focusing on too many areas or they are focused on a small number of tasks. Our method produces these unbiased and statistical measurements automatically.

The organization of the paper is as follows. Section 2 motivates our work and presents metrics that will be used to study evolution and trends in the life-cycle of software systems. Section 3 describes an overview of the code development process. We present source control systems that are used for large software projects and give an overview of the type of data stored in them. Section 4 introduces canonical sets and explains how they are used in our analysis. Section 5 details our approach and the techniques used in our analysis. Section 6 presents a case study, which explores the applicability of our approach using the PostgreSQL open source database project. Section 7 discusses related work. Section 8 presents possibilities for future work. Section 9 concludes the paper.

2 Motivation and Expected Outcomes

A software system undergoes many changes throughout its lifetime. In this paper, we study the changes applied to the source code that add new functionality, enhance current features, and fix bugs. Using canonical sets and clustering techniques we answer several questions that aid in observing the development trends of a software system and planning future development activities. The following is the list of questions:

1. How many distinct categories of work were there in a specific phase of development, or specific period of time?

2. Compared to the work completed in the previous period, what are we working on now? If we have shifted our efforts, where have they been shifted to?

3. What categories of work have been introduced in the current time period? Are these the introduction of new features or reimplementation of code? How many areas have undergone maintenance, and what are those areas?

4. How much effort has been placed in each of the categories of work of the current period? How much of this effort is maintenance and how much is new development?

The answers to these questions will help identify trends in the development process and provide a succinct overview to characterize the efforts of the development team.

Our approach permits us to identify canonical changes in partitions (periods) of the lifetime of long lived projects. These canonical changes represent the main categories of changes. Using clustering techniques, we classify all other changes in a time period as being similar to one of the identified canonical items. This classification causes the creation of "Change Clusters" that group changes of a time period into clusters of similar changes. We can then compute and study the following metrics to support a manager's inquiries:

- **Number of change clusters within a time period:** By studying the number of change clusters per time period, we can determine how many activities the development team was focused on. For example we would expect the number of change clusters to spike immediately after a release with a steady decline as it approaches the next release. This would indicate that developers are introducing new features as releases approach.

- **Number of new change clusters within a time period relative to the previous time period:** By studying the new change clusters relative to the previous time period we can identify stable areas of development. A cluster that is consistently present from one period to the next, the cluster can be inferred to be under constant maintenance.
- **Number of new change clusters within a time period relative to all prior time periods:** By studying the number and nature of new change clusters over time, we can determine when new features are being added, features are being reimplemented, or maintenance work is being performed. We would expect to see a downward trend in these numbers unless several new features are being added or many partially implemented features are being completed.
- **Distribution of changes in a time period between the identified change clusters:** By studying the distribution of changes we can see how many changes fall into each cluster and determine how much change is represented by new or old work. For example, although our analysis may reveal that a time period has a large number of distinct change clusters, by studying the distribution of changes we can recognize that the development team is performing maintenance activities and that each change cluster contains a small number of changes. Furthermore we are able to quantify not only the effort placed in each cluster, but also classify the amount of work completed as new feature implementation or maintenance.

3 Code Development Process

Source control systems are used extensively by large software projects to control and manage their source code [24, 27]; examples are RCS [27], CVS [4, 7] and Perforce [22]. These systems help coordinate the development process between various members of the team and provide the ability to restore the source code to its state at any given time in the past. For example, developers can retrieve a source code file that is no longer part of the project or roll back to a previous version of a file if they discovered that their changes are inappropriate or are too complex to maintain and understand. Furthermore, source control systems provide tools to reconcile changes made by developers working simultaneously on the same file.

The repository of a source control system usually tracks the creation, and initial content of each file. In addition, it maintains a record of every change to a file. For every change, a *modification record* stores the date of the change, the name of the developer who performed it, the specific lines that were changed (added or deleted), and a detailed explanation message entered by the developer giving the reason for the change. Using the information stored in the source control system, we can also recover change sets (files that were changed together by the same developer within short time frame). We recover for selected time periods, change lists, which are the lists of all the changes applied to a system, in the order that they were applied.

Our analysis uses the information stored in source control repositories to characterize time periods within the lifetime of a project. Our analysis also compares work done during a specific time period to work done in prior time periods. The data used in our analysis is recovered from source control systems using techniques documented elsewhere [13].

An assumption of our analysis is that each change set contains only changes that are related and focuses on a specific area of work. In principle, it is possible that a developer may "check in" several unrelated files together, though this occurs rarely. This is a reasonable assumption based on the development process employed by the studied open source projects and discussions with open source developers [2, 17, 29]. In most open source projects, access to the source code repository is limited. Only a few selected developers have permission to submit code changes to the repository. Changes are analyzed and discussed over newsgroups, email, and mail lists before they are submitted [3, 18, 31]. This review process reduces the possibility of unrelated changes being submitted together. Moreover, the review process helps ensure that changes contain all relevant files.

4 Computing Canonical Changes

Our technique for computing canonical sets is described in our previous work [25], where we presented a framework for reducing a set of features to a smaller subset called the stable bounded canonical set. In the context of change lists we refer to this subset as the canonical changes. The set of canonical changes contains changes that are as dissimilar as possible and best represent the changes not contained in that set. More formally, our notion of the canonical set is the subset with the following properties; members of the set are minimally similar, members of the set are maximally similar to changes not in the set, and members of the set are maximally stable (according to some measure).

The canonical set is a representative subset obtained through an optimization process that takes into account the structure of the relationships between the changes. The structure of the change lists is encoded into a graph where each change is represented as a vertex. Edges have weights corresponding to the similarity between their vertices as measured by the method described in Section 4.1. The

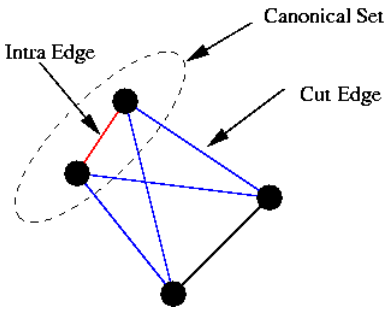

Figure 1. Canonical Set Edges: Intra edges are denoted by red, Cut edges are denoted by blue.

framework permits a weight to be associated with each vertex that indicates the relative stability of the vertex. For the work described here, we assign equal weights to all of the vertices. All changes are considered equally stable and hence equally important.

To formulate the problem, we refer to edges that have both endpoints in the canonical set as *intra* edges (See Figure 1). We refer to edges that have one endpoint in the canonical set and the other outside of it as *cut* edges. The canonical set can then be described as the subset of vertices such that the sum of the weights of the intra edges is minimized, the sum of the weights of the cut edges is maximized, and the sum of the weights of the vertices in the canonical set is maximized. Thus the canonical set is a subset of the vertices in the graph that best represents the graph with respect to the similarity and stability measures.

Graph optimization problems such as this are known to be intractable [10]. The work of Goemans and Williamson [11] on the MAX-CUT problem in graphs showed semidefinite programming (SDP) relaxations to be useful in obtaining improved approximation algorithms for several optimization problems. Using similar techniques, we formulate the canonical set problem as an integer programming problem, and then relax it to a semidefinite program. We then use an off-the-shelf SDP solver [28] to find an approximate solution. For details of this procedure the reader is referred to our previous work [20, 6, 5].

4.1 Similarity Measure

In order to compute the similarity between commits we considered the Jaccard coefficient. The Jaccard coefficient is a measurement of asymmetric information on binary variables. For our purpose we compute the coefficient as:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

where X and Y represent individual commits. That is, they are sets of files representing a single commit. We compute the similarity for each pair of commits in order to obtain the similarity matrix for the calculation of the canonical set.

4.2 Application of Canonical Sets to Change Lists

In order to gain a better understanding of how canonical sets can be applied to change lists, let us consider an example of changes applied to a software system. Given m unique changes applied to a system listed as the files altered as part of that change, we can compute a similarity of that change to all the changes that occurred during that time frame. We list these similarity measures in a $m \times m$ matrix where the diagonal is perfect similarity. Each row and column represents a different change. The measures are computed using the Jaccard coefficient.

We then compute, based on this matrix of similarity measures, the canonical changes of the period. We do not define a minimum or maximum number of changes to find, and leave that determination up to the canonical set solver. This is very important in that we are told how many changes, from the given set of changes, are needed to represent all the changes that took place during the period. It is important to note that the canonical changes are actual changes that were applied to the system.

Returning to our example of the changes applied to a software system we may find that all the changes are very dissimilar, and have no files in common. Such a situation would cause our solver to give back the entire set of changes as canonical as it would require all the changes to represent the entire set. On the other hand, if the majority were very small changes that contained only a few files in each change and one very large change that incorporated all the files of the other changes, then our solver would provide us with that one large change as it best represents the entire set of changes. However, none of these two cases is likely since similar features are implemented with common files based on practices of code re-use and modularity [21], resulting in the localization of changes into sets of files.

5 Approach

In this section we define our approach to analyze source control repositories to extract the evolution of large software systems. Using the lists of changes applied to a software system over its development life-cycle we aim to answer several questions. Finding the canonical changes helps us to answer these questions about the life-cycle of a software system.

Firstly, we wish to determine the types of changes that were made during each period of the software's development. We identify the length of the period, and obtain the changes applied to the system during each period.

As compared to the work presented by Hassan and Holt [12], we base our work on the idea that the length of different periods need not be constant. We can divide the lifetime of a software system into successive periods of time as week, month, year, or any arbitrary time frame. In the context of our approach we chose periods of 3 months, where periods are defined as development up to the time in consideration.

Following partitioning the development life cycle of the system into periods, we compute the canonical changes of the period using the Jaccard similarity measure. The canonical changes represent the main types or clusters of changes.

Furthermore, the advantage of finding the canonical changes as opposed to a histogram of files represented in each change is that the canonical changes actually represent all the changes of the period that have been applied to the system. A histogram, on the other hand, simply measures the frequency of files in each change.

It is also important to note that the number of changes applied to a system does not skew the results of the canonical changes, a very important feature of our approach. For example, consider two sets of developers working on the same features; one commits their changes to a source control repository frequently, the other does not. Assume that both groups are working on the same general areas, and the frequently committing group has 200 commits in the period, whereas the other has 50. In both cases the approach of taking the canonical changes will provide us with the same general clusters. The importance of this is even more exaggerated and likely when there are many developers working on different areas of a project in the same period with different styles of commit. Given a mix of developers, some who commit frequently, and others who do not, if we were to use the approach of a histogram we would have an inflated sense of importance on the areas that the developers who committed frequently worked on. Using canonical changes we would obtain the same change clusters regardless of the frequency of commits.

Next, using a clustering technique, we determine the distribution of the changes that were applied to the system with respect to the canonical changes of each period. That is, we classify all the changes that are not canonical in a period as being in one of the change clusters associated with a canonical change. We use the Jaccard similarity once again to determine the change clusters to which each change belongs by associating each change with its nearest neighbor in the set of canonical changes. This gives us not only the change clusters and the size of each cluster, but by proxy, a measure of effort distribution.

In order to determine if a change cluster is new, we see if the same cluster or a similar cluster has been represented in a previous period. We determine this by counting how many of the canonical changes of a period have been previously represented. For each period, we compare its canonical changes with the canonical changes of every period previous to it, and compute the canonical changes of the union of the canonical changes of those two periods. Based on the results of these period-by-period comparisons, we can determine if a change cluster has not been previously seen, indicating new feature development, or significant reimplementation of code. We refer to these changes as new change clusters over time.

Similarly, we can determine whether the focus of the development from one period to the next has changed. Not only can we observe that from one period to the next the focus of development went from maintenance to new development, but we can state that the focus of development went from maintaining a particular set of features, and developing another set of new features, to maintaining another set of features. We compute the new change clusters relative to the previous period, which indicate a stability of development, as mentioned. If from one period to the next the change clusters remain constant, the development is fairly stable; however, if they shift, and we see a significant number of new canonical changes relative to the previous period then we can assume the focus has changed. In addition, if we have an intersection of change clusters over time and relative to the previous period we observe the implementation of new features.

Based on these findings, we are able to make conclusions regarding the development life cycle of a software system as well as draw conclusions regarding it without having previous knowledge of the system. We can then go to the revision history, and corroborate this information to see what new features have been implemented. In the following section we provide a case study of PostgreSQL, where we use this approach and justify our conclusions with the development history.

6 Case Study

To demonstrate the feasibility of our approach, we analyzed the software life-cycle of the open source object-relational database, PostgreSQL. PostgreSQL is marketed as an alternative to commercial database systems such as MSSQL and DB2, as well as a more robust option to other open source systems like MySQL and Firebird. Our analysis of PostgreSQL begins with its first formal release, version 6.0. Previous to that release, PostgreSQL was referred to as Ingres.

In evaluating PostgreSQL, we chose to use periods of three months to find canonical changes. We experimented with various other time frames and found that a time frame shorter than 3 months (1-2 months) produced similar results, whereas a longer time frame did not provide sufficient information to draw conclusions about the progress of the development of PostgreSQL.

Given the list of all changes, in the form of the set of all CVS source code commits for a given time period, we computed the canonical changes. This is the set of changes such that the individual changes in that set are most dissimilar to one another, and also have the property that they are very similar to changes not in the set. In other words, the canonical changes for a period is the smallest set of changes that best characterizes all the changes of that period.

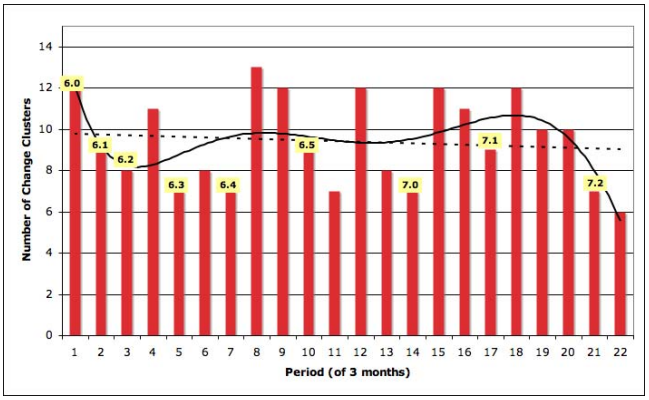

Figure 2. Number of Identified Change Clusters of PostgreSQL by Period

Figure 2 shows the number of change clusters that our approach identified for each development period in the lifetime of PostgreSQL. The figure lists the major release number for each corresponding period. The number of change clusters reveals the varying amount of change activities during each time period. We did not restrict our canonical solver to return a specific number of canonical changes. Instead, the returned canonical changes that form the basis of the change clusters are due to the inherent relationship between changes during the development process during each time period.

To uncover any overall global trends in our results, we fitted linear and polynomial (of degree 6) trendlines to our data. The linear trendline (dotted line) shows a slight downward trend. This indicates that PostgreSQL's development activity has remained active over time at a reasonably constant rate. Examining Figure 2, we note that the polynomial trendline (solid line) shows a similar trend except toward the 7.2 release where we see a decline in the number of canonical clusters. We believe this decline is likely due to missing change data for the 22nd period.

In addition to the global trends, we noticed in Figure 2 a significant rise in the number of change clusters in several periods following a release when compared to the prior period. Moreover we note a decline in the number of change clusters in a period preceding a release when compared to the prior period's. Between releases 7.0 and 7.1 or 7.1 and 7.2, we can see an increase in the number of change clusters as development commences and we note a decline as development for the release winds down. This is probably because the development of the system began with a focus on several new areas, and as the release approaches, the focus shifts to fewer and fewer areas. Intuitively, this is expected since as goals for the release are accomplished, they are no longer worked on. As a release approaches you do not add any new work, what is commonly known as a "feature freeze".

In order to examine the stability of development we can consider the number of new change clusters introduced from one period to the next. This indicates a time continuous progression of development; that is, how often the focus of development changes. It is desirable that a reasonable balance is achieved between working on old activities and pursuing new activities. This balance would ensure that development of a product progresses smoothly with new features being added and old features being maintained and enhanced.

If the development from one period to the next is very stable, and the change clusters for the two periods do not differ dramatically, this is a good indicator that the focus of development between the two periods has not changed significantly. However, if the canonical changes differ significantly, the focus of development has changed. This could be due to the introduction of new features in the software system, the updating of previous features, or the reimplementation of previous code. Our goal is to differentiate between maintenance or enhancement activities, and the introduction of new features.

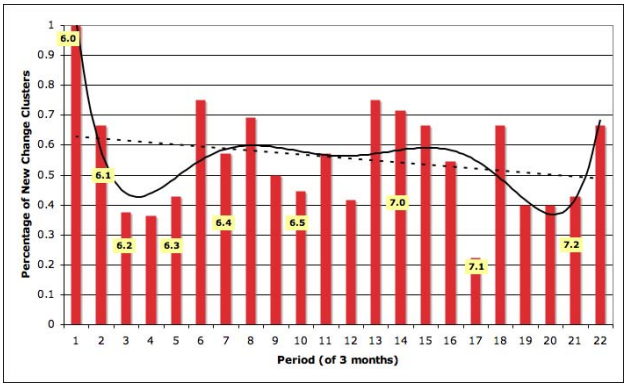

Figure 3. Percentage of New Change Clusters in a Period Relative to the Preceding Period for PostgreSQL

Figure 3 shows the percentage of new change clusters in a period relative to the preceding period. Using the same analysis as the previous figure, we fitted linear and

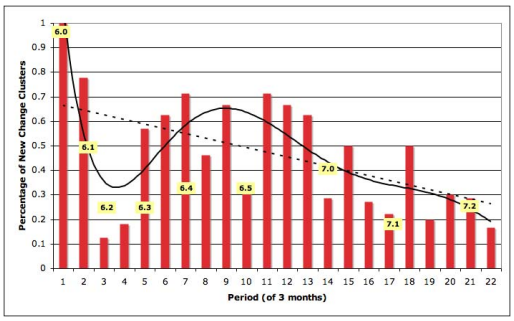

Figure 4. Number of New Change Clusters Over Time of PostgreSQL

polynomial trendlines. Both trendlines are showing a constant rate with a slight downward trend for the data. This trend indicates that the development team is working on new activities/features while continuing work on old activities/features. Moreover a closer analysis of the data reveals that for most periods following a release there is a rise in the percentage of new change clusters. Unfortunately, this trend is not consistent across all releases. Releases 6.0 and 6.1 are all done within a single period. Furthermore, the period following release 7.0 exhibits drop in the percentage of new change clusters. It is interesting to note that the periods between releases 6.5 and 7.1 (in particular periods 13-16) exhibit a rather high percentage of new change clusters when compared to the other periods in the lifetime of PostgreSQL. These few periods with a high percentage of new change clusters could be attributed to the work done in the 6.5 release to permit the team to add features more frequently. The release notes for release 6.5 support our position:

> *"This release marks a major step in the development team's mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team."*

To further corroborate this quote, we examine Figure 2 and note that periods 1 to 7 exhibited a drop in the amount of change clusters/activities that the PostgreSQL team was able to work on. Following the work done in release 6.5 the team was able to work consistently on more change clusters and was able to work on more new change clusters over time. These findings match well with the problems that the PostgreSQL team has noted (slower development progress). Their refactoring work in release 6.5 has helped them address this concern well. Clearly, the increase in the number of developers working on the project could explain some of the trends that we noticed. Nevertheless we believe that the larger number of developers is not sufficient to justify these trends since a brittle code base is likely to suffer many problems that would slow down the progress of the project. An increase in the number of developers does not imply improved efficiency in the refactoring process if the code is fragile.

In order to show conclusively that new features are actually being implemented or reimplemented (which in the context of our approach is viewed as a new feature being implemented with same or similar functionality), we see if a change cluster is considered new over the entire lifetime of the software's development. Similarly to the process of determining new change clusters relative to the previous period, we see whether a change cluster has been represented in any previous periods.

Figure 4 shows the percentage of new change clusters in a period relative to all prior periods. The trendlines in the figure show the following characteristics:

1. There is a downward trend in the addition of new features in the PostgreSQL project. This downward trend of adding new features may indicate the project's maturity over time as the focus shifts from adding new features to mainly servicing small enhancements and bug fixes [23]. The constant rate of work ($i.e.$, change clusters), shown in Figure 2, indicates that development in the project is still continuing at a constant rate but with development focus shifting into more maintenance and servicing.

2. Examining the polynomial fit (solid line), it appears that following release 7.0 the work on new change clusters have slowed down drastically. Consulting the release notes for the following releases shows that these releases focused on optimizing the database to handle large workload and on removing many limitations. To achieve these goals, the releases are likely to focus more on reworking old features and enhancing them instead of adding new features.

Once we have the change clusters for a period we can determine the distribution of changes with respect to the period's clusters. For example in period 3, there are 8 change clusters. However, it may not be that changes in these clusters are evenly distributed; in fact, it is very unlikely that is the case. In order to determine the distribution of changes between the different change clusters, we consider all the changes of the period and associate each change with it's nearest neighbor in the set of canonical changes using the similarity measure we already computed to determine those canonical changes. In period 3 we can see that the distribution of the change clusters is largely in one area (at one-third of the effort distribution) with the rest of the clusters being fairly evenly distributed. This measure in actuality is, by

proxy, a measure of the effort distribution for the period. We can see the distribution in Figure 5.

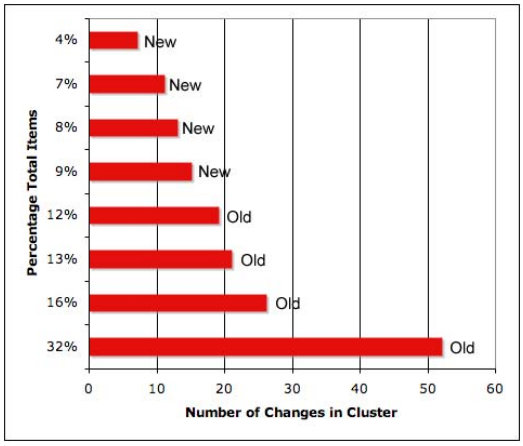

Figure 5. Distribution of Changes in Period 3 Between Identified Change Clusters of PostgreSQL

Using this information we can determine how much change occurred in a specific area. Rather than simply stating that there were 8 change clusters, we can actually state how much of the overall changes were represented in each of these 8 clusters. In the example of period 3, we can further characterize each of these clusters as new and old development using the number of new canonical changes over time. The last 4 items in the graph can be characterized as old (maintenance) and the first 4 as new development. This provides a more refined view of the status of development for a given period.

7 Related Work

The work presented in this paper analyzes historical project information stored in software repositories, such as source control repositories, to derive a characterization for different time periods throughout the lifetime of a software project. We now briefly overview previous work and contrast our approach to such work.

Works by Lehman [14], Godfrey [16], and Gall *et al.* [9] measure global software metrics such as LOC and number of changes to recognize areas and periods of rapid and slow evolution. Such approaches cannot recognize how periods relate to each other or what are the main work activities within specific periods. For example, even though a time period had thousands of changes all the changes could be mapped to mainly developing a single feature or a limited set of features.

Hassan and Holt [12] propose measuring the spread of changes over the files or subsystems of software system. Their intuition is that periods with changes spread out through the source code are good indicators of developers working on a large number of activities concurrently (*e.g.*, bugs or features) in the same time. In contrast to our presented approach, their approach does not give actual examples of the activities. Their approach simply states that too much work is being done but it cannot give example changes that characterize the different work activities. Furthermore, they do not investigate how work activities within different periods relate to each other, *i.e.*, whether work activities during a time period represent continuation of old work activities or if they represent new activities.

Lientz *et al.* [15] present the results of a survey that describes the amount of effort in large software projects allocated to the different maintenance categories such as perfective, adaptive, etc. Recent work by Schach *et al.* [26] explored the results of the survey through manual analysis of source control data. The presented approach can automatically compare the work activities across time periods in the lifetime of a software project. The approach can determine whether activities are new development or continuation of old work. The old work activities could be considered as maintenance activities. An automated lexical analysis, such as the one used by Mockus and Votta [19], can be employed to divide modifications into different categories (such as adaptive maintenance or inspection maintenance) based on the content of the detailed message attached to a modification.

Barry *et al.* [1], and Xing and Stroulia [30] characterize different time periods within a project's lifetime using a combination of characteristics about the spread of changes, the amount of changes, and the effect of the changes on the dependencies structure. In contrast to our approach, the aforementioned work simply describes a single major development focus during a time period such as rapid development, restructuring, etc. The aforementioned work is not able to identify the few main work activities in a time period such as work to implement particular features.

8 Future work

We would like to analyze more software systems using our approach in order to find more general trends. This will not only demonstrate further that our approach is effective in objectively characterizing the evolution of a software system, but also may reveal trends that we did not expect to see. Considering systems with varying longevity would allow us to determine what trends are typical for different length projects.

The canonical set framework allows for the weighting of changes by providing a saliency measure. In this work, we assumed that all changes were as important as all other

changes, which is more than likely not the case. In our future work we hope to correlate features with change sequences and provide a measure of saliency such that it is higher for features that have either been part of the system for a longer period of time, or are more critical to the system, or have been updated more frequently.

In addition, we would also like to examine systems based on varying period lengths. Rather than partitioning periods, as we did in this paper, to a fixed time frame, we would like to partition the life-cycle of a system into varied lengths. This would allow a manager to answer questions such as: "how has the development of my system evolved in the past 2 months as compared to the previous year?"

In terms of planning for future development, we would like to be able to provide managers with information so that they can anticipate not only the type of development, which would be useful, but the areas in which development will occur. This could be achieved by recognizing previous patterns of canonical changes, which would allow managers to assign tasks more appropriately to developers based on the upcoming needs of development on the system.

Lastly, it would be beneficial to testing teams for consider bug fixes and bug counts in our analysis. If we could correlate the introduction of new code and new bugs, as well as when maintenance work is primarily bug fixes, we could aid managers in assigning work tasks more appropriately.

9 Conclusions

A good understanding of the progress of code change activities in large software projects is essential to ensure that managers can monitor the progress of a project and plan for future activities. In this paper we presented an approach to characterizing the evolution of software systems and providing managers with an analysis of change activities.

The approach of using canonical sets not only allows managers to determine what change activities are being focused on during a given period, but also provides more information. Specifically, it provides managers with information about the number of areas where changes have been applied. This allows them to see when the development team has focused on several activities or just on few. It also shows managers what those activities are by providing representative examples of them.

We can also identify when code maintenance or refactoring work is being performed as compared to new development on the software system. We can then list what development occurred and when it occurred. Comparing the canonical changes of two consecutive periods provides similar information that depicts how the focus of development changes from one period to the next.

Using this information, and a clustering technique, we are able to show the distribution of changes between identified change clusters. The number of changes and the number of change clusters individually show how much change occurred in a system and what the change areas were, respectively. However, neither or them can depict the distribution of change activity. By applying a clustering technique we are able to show the distribution of changes with respect to individual change clusters, which can be inferred as a distribution of effort.

The advantages of using canonical sets to depict software evolution lie in its ability to define the number of change clusters as well as represent those change clusters by examples. It does not rely on the frequency of commits, as a histogram would, but provides the canonical changes by exploiting the structure of the changes made during a time period. It is capable of accurately depicting the change activity of a period without being skewed by the commit frequency of developers. Our approach not only avoids human intervention with its objectivity but also quantitatively characterizes trends in software systems that would otherwise require an in-depth knowledge of the entire life-cycle of the system.

References

[1] E. J. Barry, C. F. Kemere, and S. A. Slaughter. On the uniformity of software evolution patterns. In *Proceedings of the 25th International Conference on Software Engineering*, pages 106–113, Portland, Oregon, May 2003.

[2] A. Bauer and M. Pizka. The contribution of free software to software evolution. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, Sept. 2003.

[3] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–419, Portland, Oregon, May 2003.

[4] CVS - Concurrent Versions System. Available online at `http://www.cvshome.org`.

[5] T. Denton, J. Abrahamson, and A. Shokoufandeh. Approximation of canonical sets and their application to 2d view simplification. In *IEEE Conference on Computer Vision and Pattern Recognition*, Washington, DC, June 2004.

[6] T. Denton, M. F. Demirci, J. Abrahamson, A. Shokoufandeh, and S. Dickinson. Bounded canonical sets and their applications to view indexing. In *17th IAPR International Conference on Pattern Recognition*, Cambridge, United Kingdom, August 2004.

[7] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999.

[8] M. Fowler. *Refactoring: Improving the Design of Exisiting Code*. Addison Wesely Longman, Boston, USA, 1999.

[9] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, Sept. 2003.

[10] M. R. Gary and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979. (ND2,SR1).

[11] M. X. Goemans and D. P. Williamson. .878-approximation algorithms for MAX CUT and MAX 2SAT. In *Twenty-sixth Annual ACM Symposium on Theory of Computing*, pages 422–431, New York, 1994.

[12] A. E. Hassan and R. C. Holt. The Chaos of Software Development. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Helsinki, Finland, Sept. 2003.

[13] A. E. Hassan and R. C. Holt. C-REX: An Evolutionary Code Extractor for C. May 2004. Submitted for Publication.

[14] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution the nineties view. In *Proceedings of the 4th International Software Metrics Symposium*, Albuquerque, NM, 1997.

[15] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[16] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 131–142, San Jose, California, Oct. 2000.

[17] M. Mitchell. GCC 3.0 State of the Source. In *4th Annual Linux Showcase and Conference*, Atlanta, Georgia, Oct. 2000.

[18] A. Mockus, R. T. Fielding, and J. D. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 2000. ACM Press.

[19] A. Mockus and L. G. Votta. Identifying reasons for software change using historic databases. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 120–130, San Jose, California, Oct. 2000.

[20] J. Novatnack, T. Denton, A. Shokoufandeh, and L. Bretzner. Stable bounded canonical sets and image matching. In *Energy Minimization Methods in Computer Vision and Pattern Recognition, EMMCVPR 2005*, November 2005.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules.

[22] Perforce - The Fastest Software Configuration Management System. Available online at `http://www.perforce.com`.

[23] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, July 2000.

[24] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[25] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *Proceedings of the 21st International Conference on Software Maintenance*, pages 155–164. IEEE Computer Society, 2005.

[26] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt. Determining the Distribution of Maintenance Categories: Survey versus Measurement. *Empirical Software Engineering*, 8(4):351–365, Dec. 2003.

[27] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[28] K. C. Toh, M. J. Todd, and R. Tutuncu. SDPT3 — a Matlab software package for semidefinite programming. *Optimization Methods and Software*, 11:545–581, 1999.

[29] Z. Weinberg. A Maintenance Programmer's View of GCC. In *First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[30] Z. Xing and E. Stroulia. Understanding Phases and Styles of Object-Oriented Systems Evolution. pages 242–251, Chicago, USA, Sept. 2004.

[31] Y. Ye and K. Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 419–429, Portland, Oregon, May 2003. ACM Press.