College of Engineering

Drexel E-Repository and Archive (iDEA)
http://idea.library.drexel.edu/

Drexel University Libraries
www.library.drexel.edu

The following item is made available as a courtesy to scholars by the author(s) and Drexel University Library and may contain materials and content, including computer code and tags, artwork, text, graphics, images, and illustrations (Material) which may be protected by copyright law. Unless otherwise noted, the Material is made available for non profit and educational purposes, such as research, teaching and private study. For these limited purposes, you may reproduce (print, download or make copies) the Material without prior permission. All copies must include any copyright notice originally included with the Material. **You must seek permission from the authors or copyright owners for all uses that are not allowed by fair use and other provisions of the U.S. Copyright Law.** The responsibility for making an independent legal assessment and securing any necessary permission rests with persons desiring to reproduce or use the Material.

Please direct questions to archives@drexel.edu

Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence

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

Abstract

The implementations of software features evolve as an application matures. We define a measure of feature ***implementation overlap*** *that determines how similar features are in their execution by examining their call graphs. We consider how this measure changes over time, and evaluate the hypothesis that over time and subsequent versions of a software application, the implementations of semantically similar features converge. As the features of an application converge in their implementation, we are able to more effectively determine groups of semantically similar features and to reduce the cost of program comprehension by selecting few key features that give an overview of the system. We present a case study analyzing the features of the Jext, Firefox, and Gaim software systems to support our hypothesis.*

1 Introduction

In previous work [5] we developed a technique that automatically identifies the canonical features of a software application. The canonical features of a software system are a relatively small subset of features which can accurately describe all of the features of that system. By understanding the implementation of these canonical features of an application an engineer can obtain an informative summary of the implementation of the entire application. This result makes progress toward reducing the cost of understanding and maintaining similar features. We have already demonstrated the effectiveness of our technique in previous work [5] by applying it to single versions of three software systems: Jext, Firefox, and Gaim.

Each canonical feature constitutes a centroid for a cluster of features that are closely related and have similar implementations. The *implementation overlap* of two features measures the similarity of the implementations of the two features. We predict that, as a result of continual refactoring, the implementations of the features in the clusters continually become more similar to each other. That is, the implementations of features in the same feature cluster *converge* and their overlap increases over time.

The case study described in this paper supports our hypothesis that, over time and across subsequent versions of a software application, the implementations of the features in each cluster converge (*i.e.,* their implementation overlap increases).

One implication of this hypothesis is that we can use our technique to determine feature sets such that, studying a representative, canonical, feature of that set will give an understanding of the implementation of the other features in that set. Furthermore we will be able to observe how a system was refactored, and to what degree features of that system have converged. That is, we expect refactored systems to exhibit feature implementation convergence within the same feature clusters. If this property is true in general, it simplifies the task of understanding, since past knowledge is highly leveraged. Additionally, time is saved not only in comprehending features, but also in assigning the most appropriate engineers to specific maintenance tasks. For example, it is efficient to assign an engineer to a maintenance task that involves modifying a feature that is in the same cluster as another feature that the engineer has worked on in the past. According to our hypothesis, any developer with ex-

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

pertise in the implementation of any one of the features in a cluster already (transitively) possesses an understanding of all the features of that cluster since they all share implementation.

Conversely, if opportunities for refactoring were overlooked, we suspect that the implementations of semantically similar features (i.e., `Save` and `Save As` in a word processing application) may not converge over time. However, in the systems that we studied, we did not observe this phenomenon, which indicates that the developers refactored their systems properly.

The organization of the paper is as follows: Section 2 presents the process of calculating feature implementation overlap, determining the canonical set of features of an application, and evaluating the convergence of feature implementations. Section 3 present a case study on two prominent software systems, Firefox, and Gaim, evaluating the hypothesis that semantically similar features converge over subsequent versions of software. Section 4 presents related work. Conclusions and future directions of our work are presented in Section 5.

2 Technique

We next describe our approach to evaluating the implementation convergence of software features. We build upon previous work [5, 6] where we presented a framework for computing the Canonical Feature Sets (CFSs) of a software system. The tool chain shown in Figure 1 depicts the process of evaluating the convergence of an application, as well as the intermediate results of the various tools used.

2.1 Feature extraction and representation

We begin by obtaining major releases of the software system we are analyzing. For each version of the software, we independently identify its features using release notes, documentation, use cases, or the built-in help system of the software. A list of the Jext features is shown in Table 1.

Next, the features of the software are executed under the supervision of a dynamic analysis tool. This tool records the objects, functions, and variables that were exercised during the execution of the software undergoing analysis and retains this information in a call graph. For each version of the application, we obtain the call graph of each feature that was executed.

	Feature Name		*Feature Name*
1	startup	8	type
2	file-open	9	email-doc
3	file-open-doc	10	bookmark-open
4	save	11	bookmark-add
5	cut	12	search
6	copy	13	search and replace
7	paste	14	exit

Table 1: Features of Jext Text Editor.

2.2 Implementation overlap

Feature implementation overlap measures the degree of similarity between two features. It is computed by using the call graphs of two features and takes into account the number of shared method calls and the structure of the calls made (*i.e.*, call graph). Features that have similar call graphs, have a higher implementation overlap.

Intuitively, one can think of overlap as a more strict similarity measure as compared to method reuse, which only consists of the nodes of the call graph. Features exhibiting significant overlap have a high degree of method reuse; since, not only are the same methods being used between features, but also in the same pattern (See Figure 2). Alternatively, it is possible to observe a low degree of overlap and a high degree of method reuse. This indicates that although the same methods are being used between two features, they are not being used in the same way.

High method reuse without high overlap may result in an increase in the cost of understanding the software. The high method reuse implies that methods are being used in potentially different ways across the features of the system. Additionally, a bug fix in one feature may require additional effort since the methods that must be updated are not used in the same way amongst all the features.

If the features have significant implementation overlap, then the not only are methods being reused, they are being used in the same fashion in all the overlapping features. Therefore, by understanding how the methods are being reused in one feature, can provide insight into how they are being reused in other features.

For each version of the application, we compute the pairwise implementation overlap of all features. In order to measure the implementation overlap between features, we compute the association graph between the call graphs of the two features. Formally, the no-

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

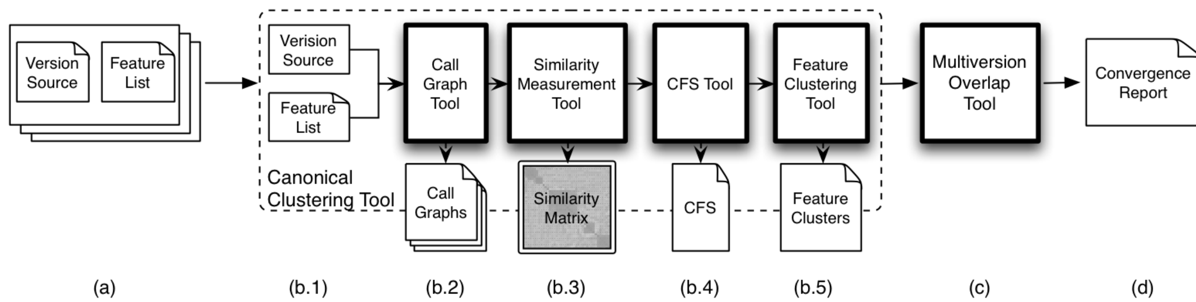

Figure 1: Tool chain; a) Multiple versions of a software system with each version's corresponding feature list; b) Canonical clustering tool, finds the canonical features and clusters the features of the system such that the centroid of each cluster is an element from the canonical feature set; c) Multiversion overlap tool, calculates the overlap in features over time, d) Convergence report, provides an evaluation of the convergence of the implementation of the features of a cluster.

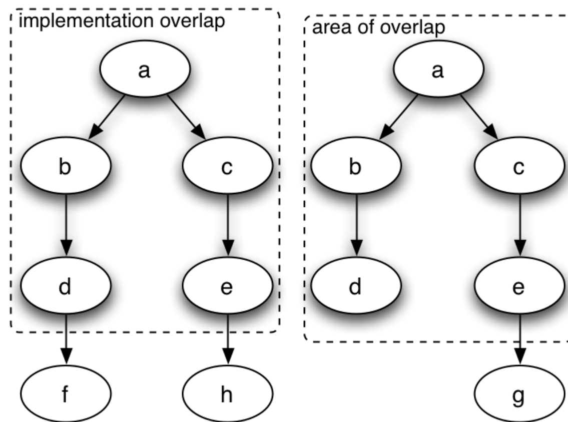

Figure 2: Two features that exhibit significant overlap and method reuse. Both features not only share a large number of methods, but the usage pattern of those methods are the same. The invocations are included in the similarity.

tion of implementation overlap of two features is the cardinality of the maximal clique in the computed association graph [11].

An association graph is constructed whose maximal cliques are in one-to-one correspondence with maximal subtree isomorphisms. The formulation allows for the mapping of hierarchical information embedded in two trees onto a flat structure. Then, using the Motzkin-Straus theorem [8] the clique problem is formulated as a continuous quadratic program. The program is then solved utilizing replicator equations, as described by Pellilo *et. al* [11]. This approach to matching hierarchical structures has been applied to various problems including those of pattern matching and object recognition.

We compute the pairwise implementation overlap of features and construct a matrix as shown in Table 2. Each value in the matrix represents the overlap in the call graphs of two features. The underlying assumption that we make are that related features in large software systems share significant amounts of code. Therefore, the dynamic call graphs that are created during the execution of two similar features should have several vertices (functions) and edges (function call relations) in common. We can justify this assumption since call graphs have been shown to depict the implementation of features accurately [9, 2]. Therefore, the implementation overlap between two call graphs is a reasonable approximation the semantic similarity of two features.

2.3 Canonical features sets

We use the implementation overlap matrix, as the similarity matrix described in Figure 1 (b.3), to compute the canonical features set (CFS) of the software system [5]. The CFS is a representative subset of features obtained through an optimization process that takes into account the structure of the relationships between the features. These relationships are encoded into a graph where each feature is represented as a vertex. Edges have weights corresponding to the similarity between their vertices as measured by their implementation overlap. A weight can be associated with each vertex that indicates the relative stability of the vertex, but in the work described here, we assign equal weights to all of the vertices. All features are considered equally stable and hence equally important.

IEEE COMPUTER SOCIETY

	startup	file-open	url-open-doc	cut	copy	paste	email-doc	bookmark-open	bookmark-add	search	search/replace	shutdown
startup	1	0.12	0.1	0.04	0.02	0.07	0.07	0.16	0.1	0	0.09	0.14
file-open	0.12	1	0.57	0.14	0.07	0.35	0.22	0.77	0.29	0.06	0.19	0.4
url-open-doc	0.1	0.57	1	0.14	0.08	0.51	0.3	0.55	0.1	0.05	0.22	0.22
cut	0.04	0.14	0.14	1	0.61	0.27	0.2	0.19	0.3	0.12	0.33	0.18
copy	0.02	0.07	0.08	0.61	1	0.22	0.12	0.14	0.23	0.09	0.14	0.1
paste	0.07	0.35	0.51	0.27	0.22	1	0.14	0.33	0.26	0.03	0.35	0.11
email-doc	0.07	0.22	0.3	0.2	0.12	0.14	1	0.38	0.37	0.08	0.34	0.2
bookmark-open	0.16	0.77	0.55	0.19	0.14	0.33	0.38	1	0.22	0.04	0.24	0.4
bookmark-add	0.01	0.29	0.1	0.3	0.23	0.26	0.37	0.22	1	0.09	0.47	0.21
search	0	0.06	0.05	0.12	0.09	0.03	0.08	0.04	0.09	1	0.19	0.04
search/replace	0.09	0.19	0.22	0.33	0.14	0.35	0.34	0.24	0.47	0.19	1	0.16
shutdown	0.14	0.4	0.22	0.18	0.1	0.11	0.2	0.4	0.21	0.04	0.16	1

Table 2: Similarity Matrix for Jext listing each feature and implementation overlap between features. The pairwise overlap between two features is based on the association graph computed using the call graphs of those features. Their overlap is defined as the cardinality of the max clique in the computed association graph.

The CFS can then be described as the subset of vertices such that the sum of the weights of the edges with both endpoints in the CFS is minimized, the sum of the weights of the cut edges (one endpoint in the CFS and the other is not) 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.

We note that graph optimization problems such as this are known to be intractable [3], and refer to our previous work [5, 6, 10], where we presented a framework for approximating the CFS.

2.4 Evaluating canonical features set

In order to determine how well the CFS represents the entire set of features for a software system, we compare the statistical relationship between individual features versus all features of a software system, and individual features versus just the canonical features of that software system. First, we build two histograms for each feature in the system as seen in Figures 3 and 4. The first histogram describes the implementation overlap for a feature with every other feature as can be seen in Figure 3 for the Jext `Startup` feature. The second histogram depicts implementation overlap for that same feature, with only the canonical features of the system; which is shown for the `Startup` feature of Jext in Figure 4.

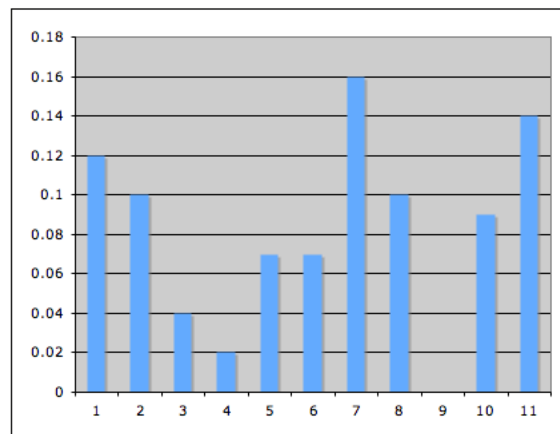

Figure 3: Histogram of the implementation overlap between the `Startup` feature and all other features of Jext. Each column shows the implementation overlap between a given feature and the `Startup` feature.

Rather than qualitatively compare the pairs of histograms for each feature, and determine that their shapes are similar, we use moments [7] which are measurements of the shape of the histograms. We

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

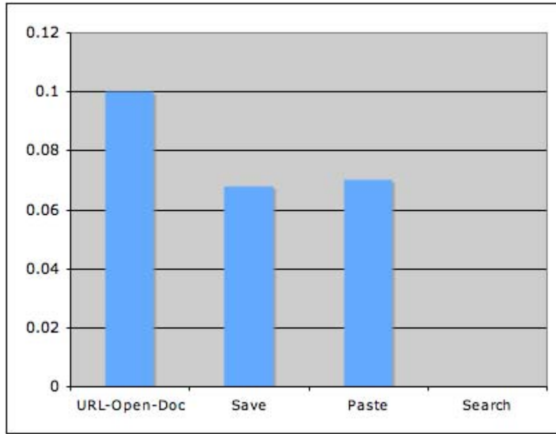

Figure 4: Histogram of the implementation overlap between the `Startup` feature and the canonical features of Jext (`URL-Open-Doc`, `Save`, `Paste`, `Search`). Each column shows the implementation overlap between a canonical feature and the `Startup` feature.

	Feature Name
1	Url-open-doc
2	Save
3	Paste
4	Search

Table 3: Canonical Feature Set (CFS) for Jext.

consider the first two moments, location and spread, to characterize the relationship between features. Location is calculated using the arithmetic mean of the data set, but is highly sensitive to changes within that data set. Spread is calculated using the variance of the data set, and is not as susceptible to changes in the data.

For each histogram we calculate these two statistical measures. We then compare the values of these measures by computing the percent error. Using these two measures, we can evaluate the CFS as a representation of the features of the software system. Table 3 shows the CFS for Jext. Table 4 depicts the moments for the histograms of the implementation overlap listed for the features of Jext. For each feature listed, we have calculated the first two moments for the histograms of all features(*i.e.*, Mean(All), Variance(All)) and the the first two moments for the histograms of just the canonical features(*i.e.*, Mean(CF), Variance(CF)). We have also calculated the error between the measures for the two histograms(*i.e.*, Error(Mean), Error(Variance)). Note that the percent error for the difference in moments is small. Since this error is small, the statistical relationship between all features is similar to that of just the canonical features, and no statistical information is lost by considering just the canonical features. Therefore, the canonical features of Jext accurately depict the system as a whole.

	Mean(All)	Mean(CF)	Percent Error(Mean)	Variance(All)	Variance(CF)	Percent Error(Variance)
startup	0.07	0.06	0.24	0.00	0.00	0.08
file->open	0.29	0.33	0.13	0.05	0.07	0.33
url->open->doc	0.26	0.28	0.08	0.04	0.11	1.72
cut	0.23	0.18	0.23	0.02	0.01	0.71
copy	0.17	0.13	0.21	0.03	0.01	0.76
save	0.13	0.10	0.23	0.02	0.04	1.06
paste	0.24	0.27	0.13	0.02	0.12	4.57
email->doc	0.22	0.17	0.21	0.01	0.01	0.01
bookmark-open	0.31	0.31	0.01	0.04	0.07	0.51
bookmark-add	0.23	0.15	0.35	0.02	0.01	0.47
search	0.07	0.04	0.44	0.00	0.00	0.92
search/replace	0.25	0.25	0.02	0.01	0.01	0.43
shutdown	0.20	0.12	0.37	0.01	0.01	0.36

Table 4: Moments of histograms representing feature implementation overlap of Jext. The moments are computed for the histogram of overlap for the feature listed (*e.g.*, `Startup`, `File-Open`, `URL-Open-Doc`) against all features, as well as the histogram of overlap for the feature listed against only the canonical features. Percent error is also listed, and indicates that simply comparing a feature to the canonical features of the system sufficiently depicts the comparison to the entire system.

2.5 Measuring implementation convergence over time

We define the notion of convergence based on the changing implementation overlap a feature has with its canonical feature, that is its nearest neighbor in the CFS, and all other features in the CFS. We have observed that features have a tendency to resemble each other in their implementation over time due to such practices as refactoring. Furthermore, as the features

IEEE COMPUTER SOCIETY

in a canonical cluster begin to converge, they begin to diverge from those not in their cluster. In other words, as newer versions of software appear, features with similar call graphs, increase in their similarity; those that have dissimilar call graphs, become more dissimilar.

Consider the overlap of a feature to the canonical features of its software system over 4 versions shown in Figure 5. The horizontal axis indicates the canonical features of the software system, whereas the vertical axis indicates the implementation overlap of the feature to the canonical feature. The overlap is shown for four versions of the software system. We observe that, over time the feature increases in overlap to its canonical feature, and decreases in overlap to all other canonical features. We can extend this to account for the fact that the canonical features each represent a subset of all of the features of the application. Therefore, the feature is becoming more similar in its implementation to all of the features in the cluster represented by its canonical feature, and more dissimilar to the other features of the software.

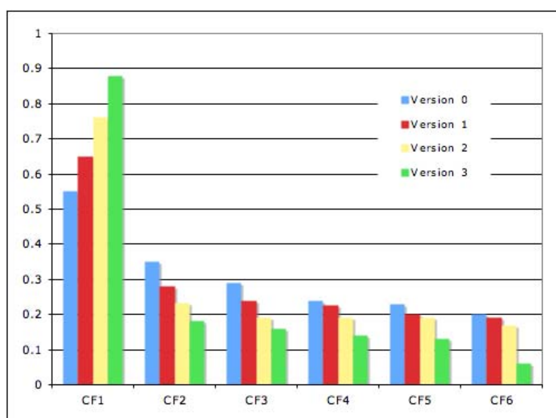

Figure 5: Feature implementation overlap evolution; This histogram presents the implementation overlap history for four versions of a software system with six canonical features. We can see that the overlap of the feature to its nearest canonical feature is monotonically increasing, whereas the overlap to other canonical features is monotonically decreasing as newer versions of the software are released.

Furthermore we define a measure of a feature's convergence with regards to the inherent structure of the system:

$$\mathcal{S}(f_k) = \frac{\epsilon(f_k, C_{f_k})}{avg(\epsilon(f_k, \{C_*\} - C_{f_k}))} \quad (1)$$

where f_k is a specific feature, $\{C_*\}$ indicates the set of all canonical features, C_{f_k} is the canonical feature with the greatest implementation overlap with f_k (*i.e.,* its canonical feature), and $\epsilon(f_k, C_{f_k})$ is the degree of overlap of f_k with its canonical feature. The term $avg(\epsilon(f_k, \{C_*\} - C_{f_k}))$ is the average overlap of feature f_k with all the canonical features except its own canonical feature. In other words, we are measuring the ratio of the degree of overlap between a single feature and its canonical feature; and the average of the overlap of that same feature with all of the other canonical features.

Observing that this measure monotonically increases for a single feature, in successive versions of the application, indicates that the feature is converging with its canonical feature, and transitively the remainder of the cluster to which it belongs. Intuitively as the development of the application continues, all the features of an application will eventually converge in implementation and resemble only those features in its canonical cluster. Examining the implementation of one feature will impart the same understanding as examining the implementation of any feature in the same cluster. Therefore, to understand the implementation of the whole system, we need only consider a select few canonical features. An expert in any one of the features of a cluster can easily understand any of the others features of the cluster to which it belongs.

3 Case Study

In order to demonstrate the effectiveness of our technique, we applied it to two prominent open-source systems: the Firefox suite, and Gaim. The Firefox suite includes a web-browser based on the Mozilla engine, and an e-mail and news client; Gaim is an Internet chat application.

As described in Section 2 we obtained multiple versions of Firefox and Gaim from the repositories listed on their websites. We also consulted the documentation from all of the versions that we obtained to determine the prominent builds of the system.

For each version of the system we determined the implementation overlap of all pairs of features by obtaining and comparing their call graphs. Using the overlaps we computed the canonical feature set of each version of the application. We also clustered the features that were not canonical with the canonical feature that had the most overlap in implementation. We then analyzed the feature overlap over the

IEEE COMPUTER SOCIETY

	Feature Name
1	File-Open Location
2	Bookmark-Add
3	Get Mail
4	Send Link
5	Edit-Find in This Page

Table 5: CFS of Firefox/Thunderbird Suite

different versions to make conclusions about the convergence of the features.

3.1 Firefox

We first applied our technique to the Firefox web-browser and email suite. Considering the 1.0.7 build of Firefox, we compute the implementation overlap matrix, and reduce the feature set of over 80 features to the canonical features show in Table 5. In order to show that this selection of features is an accurate depiction of the whole system, we compare the first two moments of the implementation overlap of every feature with all other features, as well as just the canonical features. We also compute the percent error between the value of the moments for simply the canonical features, and the moments for all features.

We observe that the percent error for the moment of the histograms representing the implementation overlap for any feature never exceeds more than fifteen percent. Furthermore the average percent error is approximately 9 percent. This indicates that for this version of Firefox, the relationship in overlap between its features can be characterized by the relationship of its features to just the canonical features of the system. The canonical features are an accurate representation of all the features of the software.

We similarly compute these moments for other versions of Firefox, and though we notice a higher percent error in earlier versions, the average percent error is consistently below twelve percent.

We then compute the method reuse in all pairs of features. Firefox exhibits a very high level of method reuse. In addition, it exhibits a very high level of implementation overlap for those features in which reuse is high. This is true for every version of the application, and it is clear why. The features of Firefox are very well centered around specific functionalities.

We observe that there are sets of features whose implementations have converged. For example we observe the clustering of features in Table 6. Intuitively, we can see that the features that are in a cluster generally have the same overall functionality. For example, `File-Open-Location` and `Bookmark-Open` both have nearly identical functionality in that they retrieve and present a webpage to the user. In terms of implementation they have converged, since they are using the same methods in an identical way. Our measure of implementation overlap effectively recognizes this similar use of methods since it is high in the case of features where the implementations and functionalities are similar.

File-Open-Location	**Bookmark-Add**
Rclick-Open in New Tab	Bookmark-Bookmark
Rclick-Open in New Window	Bookmark-Link
Go− $>*$	Rclick-Copy Link Location
File-Open File	**Get Mail**
Bookmark-Open	Read Mail
Bookmark	Get News
File-New Window	Get All Mail
File-New Tab	**Send Link**
File-Save Page	Send Message
Lclick− $>*$	Lclick-Email Address
Websearch	**Edit-Find in Page**
Startup <Link or File >	/ <Text>
	Edit-Find Again

Table 6: Clusters of features for Firefox suite; The features in **bold** fonts are the canonical features.

Nearly all of the clusters in Firefox exhibit the same implementation overlap phenomena amongst their features. The features have been isolated and have little method reuse with features outside of its cluster. In the case of the Firefox system we observe that features become isolated where they have high implementation overlap and method reuse with each other (when they are in the same cluster) or very low overlap and method reuse with those features not in the cluster.

The features of Firefox tend to converge as can be seen when comparing their overlap with the canonical features of the application. Consider the overlap of the feature `Bookmark-Open` with the canonical features of the system for four different versions shown in Figure 6. Not only is the implementation overlap to the canonical feature `Open Location` significantly greater than compared to other features, it increases in newer versions of the application. Furthermore, the implementation overlap between `Bookmark-Open` and the canonical features that are not in its cluster, is

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

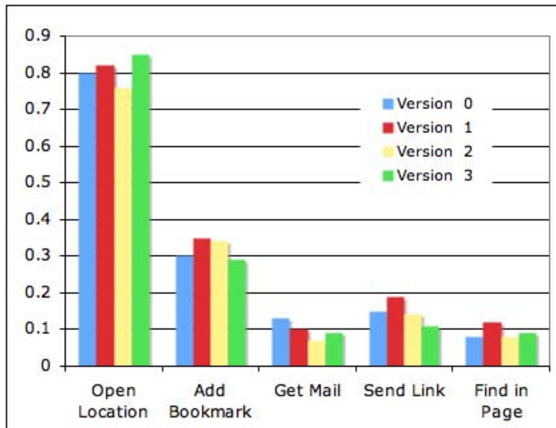

Figure 6: Implementation overlap of feature `Bookmark-Open` with the canonical features of Firefox for four different versions of the system.

significantly less, and decrease in newer versions.

The convergence can be justified quantitatively using the measure defined in Equation 1. For the initial version of the application we see that the measure is 4.85. For the successive versions the measure is 4.32, 4.83, and 5.86 for versions 1, 2, and 3, respectively. Based on this, from version 0 to 1 the convergence actually decreases, however, after that we observe a steady increase in the convergence of the feature. Similarly, the measure of convergence exhibits the same trend for the majority of features of Firefox. Averaging the convergence measure for all features of the system for each version we observe a steady monotonic increase in this value, indicating that the features of the system are converging about their canonical clusters. Furthermore, we observe that the change in the convergence of the system is greater in earlier versions than later, indicating that the convergence of Firefox is nearing realization.

We observe that because the convergence occurs across all canonical feature clusters, that the features of the system were refactored in an efficient manner such that semantically similar features share code. Since the features exhibit significant overlap within the clusters, a developer proficient with any feature of a cluster has a solid understanding of any other feature of that cluster. Also, by studying simply the canonical features of the system, we can obtain an informative understanding of the code that implements the features of the Firefox.

	Feature Name
1	Send Message MSN*
2	Send File MSN*
3	View Log
4	New Away Message AIM*
5	Add Buddy AIM*
6	Set User Info

Table 7: CFS of Gaim Instant Messaging Client. The CFS was obtained using all the features of Gaim together. An * indicated that this feature is repeated for difference protocols.

3.2 Gaim

The next application we applied our technique to was the open-source instant messaging client, Gaim. The versions of the system were obtained from the subversion repository listed on the application's page. For each version, we listed all of the features of the application, and obtained the call graphs of their execution. Using these call graphs we computed the implementation overlap of all pairs of features for each version of the application. We then, for each version, determined the subset of features that are canonical. For each version we had approximately 80 features that were reduced to a subset of six canonical features. Consider the canonical features of Gaim version 1.1 in Table 7.

To evaluate the results, we computed the moments of histograms. First we computed the first two moments of the histograms depicting each feature's implementation overlap with all other features. Next we computed the first two moments of the histograms depicting each feature's implementation overlap with only the canonical features. We computed these moments for all features of all versions. The last step in evaluating how accurately the canonical features represent the software system we computed the percent error between the moments of the histograms using just the canonical features of the systems, with the corresponding histograms using all features of the system.

The greatest error that we observe for the moments of any feature is approximately twenty percent. However, the average error is only twelve percent. This indicates that the relationship of implementation overlap between the features of Gaim can be characterized by the relationship of implementation overlap of just the canonical features. Furthermore, this indicates that if we want to study this system, studying

IEEE COMPUTER SOCIETY

the canonical features would give the best overview of the features of the system. These results are for every version of Gaim.

Similarly to Firefox, the features of Gaim exhibit a high level of method reuse, however they do not exhibit such a consistently high level of implementation overlap, although it is significant. The cases where implementation overlap is high can be attributed to the use of the graphical user interface (GUI) in the software system, and the repetitions of features for different protocols. For example, we see the same feature `Send Message` repeated for all the messaging protocols that Gaim supports, such as Oscar, Yahoo!, and MSN. There is a separate version of the `Send Message` feature for each protocol. The implementations for each of these versions however, do share the same GUI code, and therefore have high similarity, and implementation overlap since the GUI code constitutes a large portion of the code and is identical.

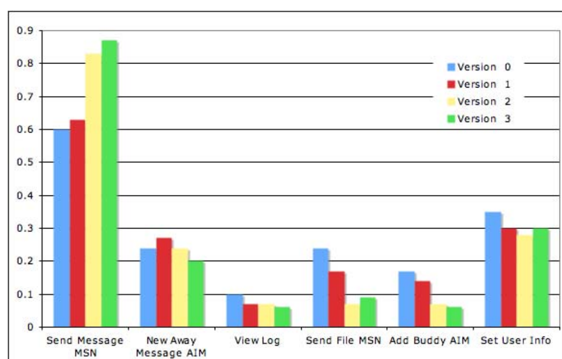

Figure 7: Implementation overlap of feature `Send Message AIM` with the canonical features of Gaim for four different versions of the system.

Consider the feature `Send Message AIM`, which sends a message over the America Online messaging service. The implementation overlap of that feature with other `Send Message` features is very high, whereas the implementation overlap and method reuse is negligible with the canonical feature `Set User Info`. The implementation overlap over four versions of the application to the canonical features of the software can be seen in Figure 7. The degree of implementation overlap with its canonical feature `Send Message MSN` increases, from the initial to final versions. However, this increase comes drastically. We attribute this to some refactoring that modularized the general `Send Message` feature, and isolated the protocol independent portions of the feature further.

Quantitatively analyzing the convergence and implementation overlap of the the feature using Equation 1, we see a steady increase of the stability of the `Send Message AIM` feature. The measure is 3.32, 3.85, 5.69, and 6.13 for versions 0 through 4, respectively. Based on this the stability of the feature is constantly increasing, and steadily. Similarly, the measure of stability exhibits the same trend for the majority of features of Gaim. Averaging the stability measure for all features of the system for each version we observe a steady monotonic increase in this value, indicating that the system is becoming more stable. Based on the design of Gaim we can attribute their stability to refinement of the features, and the isolate of protocol specific code. The usage of the methods are nearly identical for all the clusters of features, except for the protocol specific portions of the code.

The features of Gaim have steadily been refactored. We observe that the features of the system are split into two parts each; the common part of the feature, and the protocol specific part. The developers of the messaging client practiced information hiding effectively. If the specifications of any of these messaging protocols were to change the other protocols would not be affected. Furthermore, if they decide that they want to modify the messaging feature of the application they only need to do so in one central place.

4 Related Work

In previous work [5] we developed a framework for studying and characterizing software systems by instrumenting code to obtain the call graphs of features, and analyzing them to reduce the set of all features to a small subset that is best representative of that set. In this work we se that framework to determine the convergence of software features.

We present the notion of implementation overlap, which determines how similarly two features use methods. In other work [12] a framework of program slicing based coupling measures to evaluate software quality is presented. The framework uses well established coupling measures with slicing based source code analysis to determine software quality.

In the instances where we find many features overlapping with one another, we suggest that they be candidates for refactoring since the methods that they use are not only reused, but reused in a similar fashion. Davey *et. al.* [1] describes a technique that investigates the necessity of re-modularizing legacy code.

IEEE COMPUTER SOCIETY

They consider data cohesion as an influence to remodularization and compare it with call structure.

Lastly, we study the evolution of a software system by considering the change in the implementation overlap of its features over time. Hsi and Potts [4] derive several graphical views that illustrate feature evolution. One of the views they present is a "feature clump" view which groups features with related functionality together.

5 Conclusions and future work

This work contributes to the state-of-the-art in software understanding research by providing developers within an approach to:

- characterize the overlap in implementations of software features
- determine the subset of features, called the canonical feature set in order to cluster features
- evaluate the effectiveness of the canonical feature set in representing the software system
- and quantify the convergence software features over time

Software engineers can study applications with many features by simply studying those representative canonical features, and use those features in analyzing the aspects of the software system. We demonstrate, by using the moments of histograms, that considering the relationship of features using solely the canonical features of a software system is comparable to using all of the features. Furthermore, using the canonical features we are able to recognize whether features' implementations are converging over time.

Studying Firefox and Gaim, we demonstrate not only the effectiveness of our approach but that both systems are well designed and have been refactored. We are able to justify our results based on the development of those systems.

We plan to continue working on the subject of this paper by:

- evaluating of the effectiveness of considering canonical feature sets compared to other feature reduction techniques
- performing a case study involving other diverse applications that have a large number of unique features
- studying in the change in canonical features over time (*i.e.*, how does the architecture of the software system change as newer versions are released) of software systems.

References

[1] J. Davey and E. Burd. Evaluating the suitability of data clustering for softwareremodularisation. *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 268–276, 2000.

[2] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3):210–224, 2003.

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

[4] I. Hsi and C. Potts. Studying the Evolution and Enhancement of Software Features. *Proceedings of the 2000 IEEE International Conference on Software Maintenance*, pages 143–151, 2000.

[5] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006, Benevento, October 23-27),*. IEEE Computer Society, 2006.

[6] J. Kothari, T. Denton, A. Shokoufandeh, S. Mancoridis, and A. E. Hassan. Studying the evolution of sofware systems using change clusters. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006, Athens, June 14-16),*. IEEE Computer Society, 2006.

[7] M. Mandal, T. Aboulnasr, and S. Panchanathan. Image indexing using moments an wavelets. In *IEEE Transactions on Consumer Electronics*, pages 557–565, Rosemont, IL, USA, 1996. IEEE.

[8] T. Motzkin and E. Straus. Maxima for graphs and a new proof of theorem of turan. *Canadian Journal of Mathematics*, 17:533–540, 1965.

[9] G. Murphy and D. Notkin. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '95)*, 1995.

[10] 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)*, pages 316–331, November 2005.

[11] M. Pelillo, K. Siddiqi, and S. W. Zucker. Matching hierarchical structures using association graphs. *Lecture Notes in Computer Science*, 1407, 1998.

[12] J. Rilling, W. Meng, and O. Ormandjieva. Context driven slicing based coupling measures. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004.

0-7695-2860-0/07 $20.00

© 2007 IEEE

IEEE COMPUTER SOCIETY