On Computing the Canonical Features of Software Systems

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

Abstract

Software applications typically have many features that vary in their similarity. We define a measurement of similarity between pairs of features based on their underlying implementations and use this measurement to compute a set of canonical features. The ***Canonical Features Set (CFS)*** *consists of a small number of features that are as dissimilar as possible to each other, yet are most representative of the features that are not in the CFS. The members of the CFS are distinguishing features and understanding their implementation provides the engineer with an overview of the system undergoing scrutiny. The members of the CFS can also be used as cluster centroids to partition the entire set of features. Partitioning the set of features can simplify the understanding of large and complex software systems. Additionally, when a specific feature must undergo maintenance, it is helpful to know which features are most closely related to it. We demonstrate the utility of our method through the analysis of the Jext, Firefox, and Gaim software systems.*

1 Introduction

One way of understanding a software system is to identify and comprehend its features and the code that implements those features. For complex applications the number of features may be significant, and indeed overwhelming. The implication is that in order to understand a software system, all of its features must be understood. However, it is likely that there are similarities between features. Therefore, understanding one feature assists in understanding similar features. For example, consider an application with the following features: " `Search,`" " `Save,`" and " `Save As`". One expects the " `Save`" and " `Save As`" features to be more similar (share more of their implementation source code) than the " `Search`" and " `Save`" features. This implies that comprehending " `Save`" simplifies the task of comprehending " `Save As`", and vice versa. We envision developers using this technique to comprehend a software system by considering only a few of its features to which the rest are similar.

We have developed a measure of similarity between pairs of software features that aids in partitioning them into sets. We use this measure to determine the set of canonical features of a software system. The canonical features set (CFS) consists of a small number of features, relative to the total number of features, that are most representative of the software system. Intuitively, the CFS is a set of distinguished features that characterizes a software system succinctly.

Engineers can obtain an overview of a system's capabilities by studying the features in its CFS. This follows from the fact that all other features of the system are similar to features in the CFS. Because members of the CFS are as dissimilar as possible, engineers can obtain a broad overview of the system by inspecting a small number of its features.

Engineers can also obtain an overview of a system's implementation by studying how members of its CFS have been implemented. Two features that have similar functionality should share code and thus only one of the two features should appear in the CFS. Transitively, if many

	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.

features are similar to each other, as we would expect in a large system where features share code, only one of those features should appear in the CFS.

The features in the CFS act as central points for all of the features and induces a clustering that partitions the features of a software system. Each of the clusters is anchored by a canonical feature and contains feature elements that are similar to the canonical feature.

Partitioning the set of features can reduce the cost of understanding large and complex software systems. When an engineer knows that a specific feature must undergo maintenance, understanding which features are most closely related is helpful since related features most often share significant portions of their code. Our method partitions the features of a software system around a highly representative subset of those features. Understanding this highly representative subset leverages the engineer's time investment by presenting a minimal number of specific features that reveal the most about the system.

2 Technique

In this section we describe our technique for computing Canonical Feature Sets (CFSs) and Feature Partitions (FPs) of software systems. The work-flow and tool chain is depicted in Figure 1. We explain our technique using an analysis of the Jext programmer's editor [1].

We first identify the features of the Jext program using documentation, use cases, test cases, or other means such as the help system of the software. A list of the Jext features is shown in Table 1.

Next, a set of test cases is designed to exercise the features of the program. The features are executed under the supervision of a dynamic analysis tool that captures the objects, functions, and variables that were involved. If a test suite is present for the software, we can utilize the test cases corresponding to individual features, rather than manually exercising the features. In analyzing Jext we obtained function call information using the dynamic analysis tool EJP [17].

The test cases are used with the call graph tool (Figure 1a) to produce a set of call graphs, one for each executed feature of the software being studied. Using the similarity measurement tool (Figure 1b), we next compute the pairwise similarity between the call graphs that were generated in the previous step and create a similarity matrix. Since call graphs are a direct representation of feature implementation, the similarity between two call graphs is equivalent to the similarity of the features being represented by those call graphs. Call graph similarity can be measured using simple metrics such as (a) the number of function nodes the call graphs have in common, (b) the number of call edges they have in common, such as the Jaccard [8] similarity, or (c) a more sophisticated approximate graph matching algorithm.

In order to determine the similarity between two features, we compute the association graph based on the call graphs of those features. We define their similarity as the cardinality of the maximal clique in the computed association graph [13]. We refer to this measure as the association graph matching similarity measure (AGM). We computed the AGM for all pairs of features.

As formulated by Pelillo *et. al* [13], we use a formal approach for matching hierarchical structures to determine the AGM. 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 [10] the clique problem is formulated as a continuous quadratic program. The program is then solved utilizing replicator equations, as described by Pellilo *et. al*. This approach to matching hierarchical structures has been applied to various problems including those of pattern matching and object recognition. We use this approach by considering two call graphs to build an association graph based on them. Then we determine the cardinality of the maximal clique within the computed

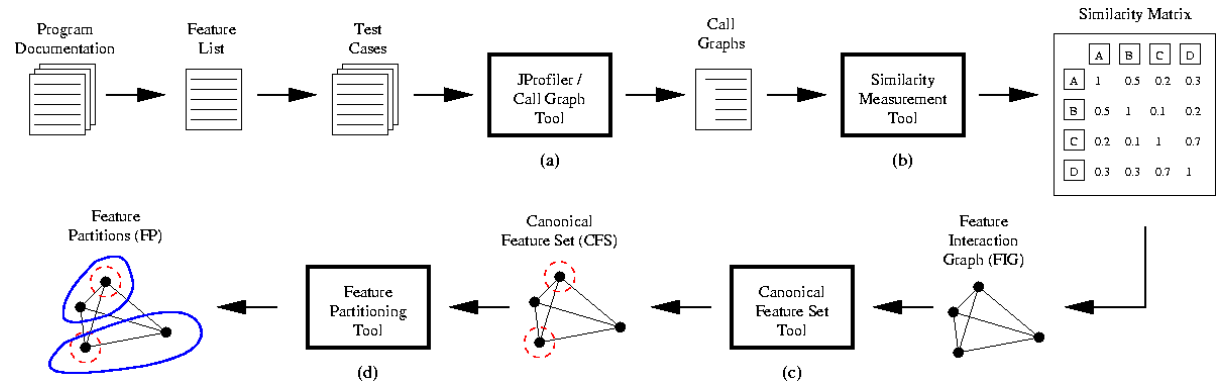

Figure 1: Work-flow and tool chain; a) Call graph tool, b) Similarity measurement tool, c) Canonical feature set tool, d) Feature partitioning tool.

association graph and use it as the measure of similarity between the two call graphs.

The underlying assumption that we make is that the implementations of similar features have a significant amount of code in common. 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 accurately depict the implementation of features [11, 4]. Table 2 shows the similarity relationships between all pairs of features for Jext.

The similarity matrix describes a graph called the feature interaction graph (FIG), whose vertices represent features of the program and whose weighted edges represent the degree of similarity between the features that are incident to the edges [15]. Feature similarity is based on the pairwise similarity of the underlying call graphs of the features. Figure 2 illustrates the FIG for Jext.

The CFS tool (Figure 1c), uses the FIG to determine the canonical feature set, which in the case of Jext is "`url-open-doc, save, paste, search`". Table 3 shows the CFS for Jext. The Canonical Feature Set (CFC) tool extracts a subset of features with the following properties.

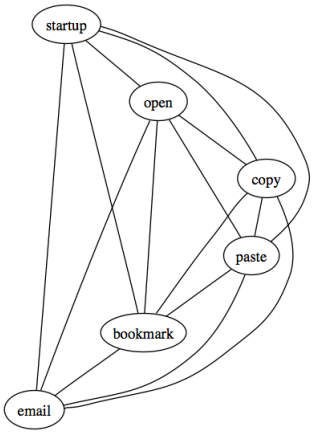

Figure 2: Partial feature interaction graph (FIG) for Jext. Each vertex in the graph represents a feature of the system. Edges between vertices represent the similarity between the features that are represented by those vertices.

I. Features in the CFS are minimally similar.
II. Features outside the CFS are maximally similar to features in the CFS.
III. Features in the CFS are maximally stable.

In previous work [12, 9, 14], we presented a framework for reducing a set of features to a smaller, more stable subset, called the stable bounded canonical set, which in this

	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 similarity between features. The pairwise similarity between two features is based on the association graph computed using the call graphs of those features. Their similarity is defined as the cardinality of the max clique in the computed association graph.

context we call the canonical feature set (CFS). The stability of a feature is a measure of the relative desirability of having a feature in the CFS; features with higher stability are preferred to be in the CFS over those with lower stability.

The input to the CFS tool consists of a set of features $\mathcal{P} = \{p_1, ..., p_n\}$, an associated set of stability measures $\tau = \{t_1, ..., t_n\}$, and a similarity function $\mathcal{W} : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{R}^+$. In our work, $\mathcal{P}$ represents the set of software features; the stability, τ, is set to one for every feature indicating no bias towards certain features in the CFS; and $\mathcal{W}$ represents the feature similarity. Given the input $\mathcal{P}, \tau$ and $\mathcal{W}$, the CFS tool computes the CFS, $\mathcal{P}^* \subseteq \mathcal{P}$. We note that the CFS is an approximate solution to a problem which has been shown to be intractable [7].

Lastly, the feature partitioning tool (Figure 1d), is used to cluster the remaining, non-canonical features, using the canonical features as partition representatives. To partition the features our technique places every non-canonical feature in a set with its nearest neighbor in the CFS. Clusters represent sets of similar features based on implementation. With an understanding of the functionality of features we can expect that features with similar functionality, share implementation, and should be placed into the same cluster. Instances where this is not the case, are anomalies and deserve further scrutiny.

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

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

Furthermore, we can observe anomalies in the implementation of features with similar functionality by examining the similarity matrix. For example, in earlier versions of Jext, we observed very little similarity between the " `Search`" and " `Search and Replace`" features. Based on their similar functionality we expected that these two features should be implemented using

shared code. After inspecting their implementations it was clear that both features were implemented separately, which explains the anomaly. In later versions we saw that the similarity between these two features increased significantly as well as the feature similarity of " `Copy`" and " `Paste`", indicating the use of common code. Inspecting the call-graphs of each of these features and their implementations corroborated the change.

	Feature Name		
1	Startup/Shutdown	7	Open/Close Tab
2	Open Location/File	8	Get/Read Mail/News
3	Save Location/File	9	Compose/Send Mail
4	Send Link	10	Block Pop-up
5	Go Link/Forward/Back	11	Find
6	Bookmark Add/Open	12	Mouse Click Functions

Table 4: Major Features of Firefox and Thunderbird

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.

We applied our technique to each application in our case study in order to analyze the features of those applications. Using the profiling methods described in Section 2 we determined the features of the system and obtained call graphs for those features. Next, we computed the pairwise similarity for all pairs of features and used those similarities in order to obtain the CFS for the application. Using the CFS and the similarities between features we partitioned the features around the elements in the CFS. Lastly, we analyzed and corroborated the results of the CFS and partitions through manual inspection of the code, and an understanding of functionality.

3.1 Firefox Suite

We first applied our technique to the Mozilla-based web-browser, Firefox, and its companion mail and news client, Thunderbird. The open-source, multi-platform browser that accounts for over of 19% of the Browser market share [18]. The browser provides features such as an integrated pop-up blocker, tabbed browsing, built-in search, live bookmarks, themes, and the ability to apply extensions to the browser for added custom functionality [5]. Since Firefox and Thunderbird share a code-base, we treated them as a single integrated system.

The versions of Firefox and Thunderbird that we considered in this analysis were 1.0.6 and 1.0.7 respectively, the two latest releases at the time. Both systems are primarily implemented with C and C++ code. To gauge the size of the system, and implicitly the complexity of its source code, we note that the Firefox suite is implemented using over 3 million lines of code and 10,000 source files, not including the shared libraries. The task of understanding a system of this size by examining its code is daunting.

We began by developing use-cases to invoke the functionality of the browser and mail reader. We also developed a variety of use-cases that employ different methods of invoking the same functionality. For example, visiting a website by typing the URL in the address bar, navigating to the "`File`" menu and choosing the " `Open Location`" option, right clicking on a particular link and selecting " `Open`", or simply left clicking on a link all result in the same functionality being invoked. Additionally, in order to consider the " `Pop-up Blocker`" as a feature, we opened a number of websites known to have pop-ups. We did this for varied number of pop-ups per site. Table 4 lists the major sets of use-cases developed. Ultimately, we used more than 80 use-cases in our analysis.

We computed the similarity between each of these use-cases using two different similarity functions. The first was the Jaccard Similarity Coefficient [8], and the second was the association graph matching similarity measure (AGM) which uses the max clique cardinality of an association graph [13]. The Jaccard coefficient we used was based on the caller-callee relationships of the methods invoked for each use-case, whereas AGM uses the execution traces (call graphs) of each use-case to obtain a similarity. In our analysis of the Firefox Suite and Gaim we used AGM. In investigating these two approaches we found that AGM is a more informed measure than the Jaccard Similarity Coefficient though it is more computationally intensive. Our motivation for using AGM was based

on the observation that Jaccard similarity does not incorporate the topography, and hence relationships, between function call orderings embedded in the call graphs. AGM builds the association graph of two execution traces and uses the cardinality of the max clique within that association graph as the measure of similarity. Figure 3 shows a heat map representation of the similarity matrix for the Firefox Suite.

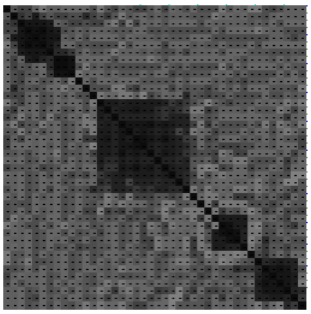

Figure 3: This image depicts a heat map of the similarity matrix for the Firefox Suite. The degree of similarity is encoded as temperature in the map. Inspection reveals areas where the temperature becomes warmer (darker) indicating high levels of similarity. The rows and columns of the similarity matrix used to make the heat map have been reordered to emphasize hot spots. To obtain the ordering, we evenly distributed members of the CFS along the rows and columns of the similarity matrix and permuted the remaining rows and columns around their corresponding nearest neighbor in the CFS.

After computing the pairwise similarity between all of the use-cases, we computed the Canonical Features Set. The CFS we computed for the Firefox suite can be seen in Table 5. The CFS does not explicitly state that browsing is a canonical feature of the suite. However, inspecting the similarities of features we see that "`Open Location in Tab`", "`Open Location in Window`", and "`Go to Location`" are all very similar to "`File-Open Location`". Any of these features could represent the group of features containing those, and similar features since they all represent the browsing functionality.

To cluster the remaining features that are not in the CFS we associate each of them with their nearest neighbor in the CFS. This induces a clustering of features where the centroids are CFS members. Table 6 shows the clustering of features using the nearest neighbor algorithm.

	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

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

In order to justify why certain features appear in the CFS we consider the release notes of the Firefox and Thunderbird applications, as well as their implementations. The first surprising result is that "`Send Link`" is in the CFS and "`Send Message`" is not. This is because "`Send Link`" subsumes the functionality of "`Send Message`" as well as places the link to be sent in the message being written. "`Send Link`" opens an email composition containing the link as the body of the message. In actuality the "`Send Link`" feature invokes a mailer command that does not necessarily need to be handled by Thunderbird, but by the default mail client. Similarly, "`Left Click on email address`" executes the same functionality of creating an email message within the Firefox suite.

It was surprising to find that many features of Firefox

are highly similar to "File-Open Location", particularly "Print" and "Save Page". Viewing a page, or opening a location or file, essentially opens a connection to a site, makes a local copy of the files needed for viewing the page in the local machine's cache, and displays the page accordingly. " Save Page" performs the exact same function as viewing a page, but does not store to the cache; rather it saves a copy to the location specified by the user. The encompassing functionality explains why the feature " File-Open Location" is in the CFS and " File-Open File" is not. "File-Open Location" shares functionality with "File-Open File" and also all the other features in the same cluster. "File-Open File" does not necessarily have as much in common with the other features in the cluster. In other words, "File-Open Location" better represents the cluster.

"Save Page" and "Print Page" have a considerably high similarity as well. After comparing the call graphs of the two features we were able to justify why they were similar. Both features load a page in the same manner, but "Save Page" simply copies the related files in the cache to the location on the disk the user specifies. However, "Print Page" writes a file to a temporary location on the disk, prints the page, and subsequently removes the file.

We expected the "New Tab" and "New Window" features to be similar, but our results showed otherwise. Taking a closer look at their functionality provided insight as to why these two features were not as similar as we expected. "New Window" will open a new window, and load a specified page, making it very similar to "File-Open Location", and "Startup" which exhibit the same behavior. "New Tab" on the other hand, opens a new tab without loading any content. Adjusting the browser's settings to open a homepage whenever a new tab was created invoked full functionality of "New Tab" and increased its similarity to "New Window".

The newly added feature of "Websearch" in the toolbar is simply a wrapper to the "Open-Link" feature, hence their high similarity. Considering the implementation we see that the text that is typed into the websearch bar is fed into a hard-coded link for the websearch engine of the user's choice and then the "Open-Location" is executed.

The features in the cluster represented by the canonical feature "Bookmark-Add" are all very similar in their implementation. The differences between them are in how they are invoked, although they ultimately yield the same functionality. One feature in the cluster, "Right Click-Copy Link Location", was not very similar to the other features in the cluster. This feature shares the functionality of retrieving the link's location. "Right Click-Copy Link Location" falls into the cluster represented by "Bookmark-Add" because it is more similar than to the other features in the CFS. The features in the clusters do not necessarily have high similarity to each other, just to the centroids.

The "Edit-Find in Page" and the "/ Text" features are nearly identical as is the "Find-Again" feature. Both the "/ Text" feature and "Edit-Find in Page" features use the same interface. The only difference in their implementation is how the feature is invoked. One is invoked by typing a "/" and the other by clicking "Edit-Find in Page".

The "Pop-up blocker" feature was also interesting as it did not seem to be invoked explicitly. After trying to understand how the feature worked, we simply created a site that would cause pop-ups to come up nonstop. At first this did not seem to do anything either. The blocker feature would be invoked once per page. We then created a series of pages that opened a pop-up at another address. It turns out the pop-up blocker allows a site, and stops checking for pop-ups once you have designated it as an acceptable site for pop-ups. Otherwise the pop-up blocking feature is always on, and checks to see if each page is causing a pop-up. If it is not, the user never actual sees the feature. The pop-up blocking feature is embedded into the browser.

Thunderbird has a few features, such as "Get POP3 Mail" and "Get IMAP Mail", that are all quite similar. They retrieve information from remote locations and store it on the local system in the same manner. The only difference is that they each require unique protocols to access and retrieve the data from the remote locations. Since the code that deals with the protocol issues is a very small part of the actual code of the features, it does not heavily impact their similarity.

3.2 Gaim

The next application that we applied our technique to was the instant messaging client, Gaim. Gaim is a client, that supports numerous protocols and is available for many operating systems including Linux, BSD, OS X, and Windows. The protocols supported with a standard distribution of Gaim are Oscar (AIM and ICQ), MSN Messenger, Yahoo!, Zephyr, IRC, Jabber, Gadu-Gadu, SILC, and Groupwise Messenger. For the purposes of this case-study we only include features for Oscar, MSN, and Yahoo!. Gaim allows users to log into several accounts using the various networks simultaneously and provides features such as file transfer, away messages, buddy pounces, theme support, and plug-in support.

Gaim proved to be an interesting case-study as it provides many similar features, for specific protocols. For example, the feature of sending a message is very similar in perceived functionality regardless of protocol (whether you are using MSN or AIM). However, we suspected that the implementation must be different since the protocols are different. In determining how to continue with the study of Gaim, we chose to use all the features of Gaim for all of the protocols. This revealed interesting design patterns used in the application.

We began by listing all the features of the system. The major features for the messenger are listed in Table 7. Features that are repeated have similar functionality for different protocols and are treated as separate features. For example, checking a buddy's status on AIM and checking a buddy's status on Yahoo! are considered to be two different features. Table 7 however, lists the feature only once, indicating that it is supported for multiple protocols with an *. Also, certain features had to be invoked with an implicit command. For example, the feature "`Receive Message`" cannot be explicitly invoked by a user. However, another user sending a message to the client application we are profiling, would invoke the feature. These features are within parenthesis in the table.

We collected the execution traces for each invoked feature. There were approximately 80 features profiled. We then computed the pairwise similarity between all of the features; Figure 4 depicts the heat map representation of the similarity matrix of Gaim.

Using the similarity matrix, we computed the CFS for Gaim as can be seen in Table 8. Each repeated feature, one that occurs in multiple protocols, was only found at most once in the CFS. After taking a closer look at the feature implementations and the design documentation we can see that all repeated features are implemented with the same code except for a small section of code that is specific to the protocol being used. This can even be seen in the GUI of Gaim as the entry points to the features are the same regardless of protocol.

For example, in the "`Send Message`" feature, which can be invoked by simply double clicking on a user in a buddy list, or navigating to the "`File`" menu, choosing "`Send Message`", and typing a contact's name will

	Feature Name		*Feature Name*
1	Startup/Shutdown	11	Add Buddy*
2	Login*/Logout*	12	Add Group*
3	Send Message*	13	Add Chat*
4	(Receive Message)*	14	Set Away Message*
5	Send File*	15	New Away Message*
6	(Receive File)*	16	Return from Away
7	Direct Message*	17	Set Buddy Pounce*
8	Accept Direct Message*	18	(Execute Buddy Pounce)*
9	View Log	19	Set User Info*
10	Get Info*		

Table 7: Major Features of Gaim Instant Messaging Client. An * indicates that this feature is repeated for different protocols. () indicates that this feature is not user-invokable.

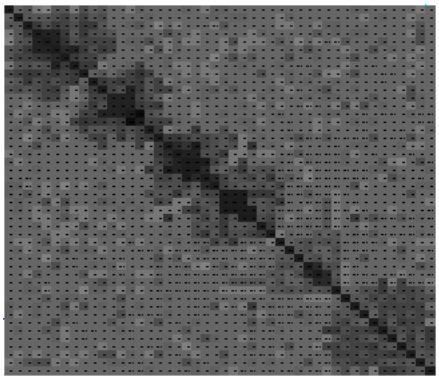

Figure 4: This image depicts a heat map of the similarity matrix for the Gaim Messaging client.

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

Table 8: 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.

perform the same actions of sending the message regardless of the protocol being used. Despite which protocol is being used, the same instructions are used for retrieving the message from the GUI, and for parsing the message, and contact name. The only difference occurs when the message is finally being sent.

4 Related Work

In previous work [16], we used dynamic analysis to gain an understanding of the Mozilla web browser by extracting various structural views of the software. In this paper, we instrument code in order to obtain function call graphs. Bruegge *et al.* describe a framework [2] to monitor runtime information such as function calls and variable modification. In their system, they instrument the source code of applications to extract this information.

Our goals in this work were to analyze features and compare their implementations to those of other features. For our purposes, we define a feature as a piece of functionality that can be invoked by a user of an application; a usage scenario [3, 4]. In other words a feature is an instance of execution of an application, which can be implemented as a test case that invokes certain behaviors of an application.

Work by Fuscher *et. al* [6] introduces an approach to identify how certain features in an application are implemented using execution traces and queries during runtime. Since execution traces reflect the actual implementation of a software system, they use them to analyze and compare different versions of the same software system in order to provide insights to its evolution.

We treat the obtained execution trace of each feature as a unique signature of that feature and identification of how it is implemented. Using graph matching techniques, we compare the signatures and implementations of these features. In previous work [13] a formal approach for matching hierarchically organized graphs is presented. They achieve this by constructing an association graph whose maximal cliques are in one-to-one correspondence with maximal subtree isomorphisms. They show that their approach is applicable to a variety of domains and is particularly effective for shape matching in computer vision. We apply this procedure in order to find the similarity between the call graphs of two features.

Elsewhere [12] we presented a framework for reducing a set of items with measurable similarity to a subset such that the subset is best representative of the entire set. Using that framework, we tracked the evolution of software systems [9] by encoding the similarity between sets of changes applied to a software system in order to determine the representative changes to that system. In this paper, we apply the same technique, but instead use the similarity between features to find the representative features.

5 Conclusions and future work

This work contributes to the state-of-the-research in software engineering by creating techniques and software tools that help software engineers understand a complex software by:

- characterizing the similarity of software features based on the similarity of their underlying implementation (*i.e.,* call-graphs);
- using this characterization of feature similarity to identify a small set of the system's canonical features that are distinguishing and representative;
- using canonical features to partition the set of features so that similar features are clustered together while keeping the clusters distinct.

The work contributes to the state-of-the-practice in software engineering by providing software engineers

with tools that can assist them in either a broad or a targeted study of a software system. By examining each canonical feature and the classification of features, software engineers can get a broad overview of the distinguishing features of the software. This is especially helpful in the absence of high-quality software documentation. Similarly, by examining a cluster of the partitioned feature set and by examining the similarity between the features in the same cluster, the software engineer can perform a systematic and targeted study of a distinct set of software capabilities.

Our plan is to continue working on the subject of this paper. Specifically, we would like to perform:

- a systematic comparison of several call-graph matching algorithms to evaluate their relative effectiveness in computing feature similarity;
- a case study that includes more software systems, especially ones that have a very large number of features (*e.g.,* Gimp);
- a mining of versioned software repositories to observe the evolution of the canonical features and canonical feature sets of the software systems in the case study.

References

[1] *Jext: Source code editor*. http://www.jext.org/.

[2] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSLA93)*, pages 65–82, 1993.

[3] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of The 17th IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 602–611, 2001.

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

[5] Firefox - Rediscover the Web, Firefox Homepage. http://www.mozilla.com/firefox. May, 2006.

[6] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind. System evolution tracking through execution trace analysis. In *IWPC*, pages 237–246. IEEE Computer Society, 2005.

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

[8] Jaccard Index. http://en.wikipedia.org/wiki/Jaccard_index. May, 2006.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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 (ICSM 2005, Budapest,September 25-30)*, pages 155–164, 2005.

[15] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.

[16] M. Salah, S. Mancoridis, G. Antoniol, and M. D. Penta. Towards employing use-cases and dynamic analysis to comprehend mozilla. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005, Budapest,September 25-30)*, pages 639–642, 2005.

[17] S. Vauclair. Extensible java profiler. Masters thesis, EPF Lausanne, 2003. http://ejp.sourceforge.net.

[18] W3 Schools. http://www.w3schools.com. May, 2006.