Towards Employing Use-cases and Dynamic Analysis to Comprehend Mozilla

Maher Salah and Spiros Mancoridis
Department of Computer Science
Drexel University
3141 Chestnut Street, Philadelphia,
PA 19104, USA
{msalah,spiros}@cs.drexel.edu

Giuliano Antoniol and Massimiliano Di Penta
Department of Engineering
University of Sannio,
Palazzo ex Poste,
Via Traiano 82100 Benevento, Italy
antoniol@ieee.org, dipenta@unisannio.it

Abstract

This paper presents an approach for comprehending large software systems using views that are created by subjecting the software systems to dynamic analysis under various use-case scenarios. Two sets of views are built from the runtime data: (1) graphs that capture the parts of the software's architecture that pertain to the use-cases; and (2) metrics that measure the intricacy of the software and the similarity between the software's use-cases. The Mozilla web browser was chosen as the subject software system in our case study due to its size, intricacy, and ability to expose the challenges of analyzing large systems.

1 Introduction

The identification of software features and of their traceability links to source code constitutes a relevant task for the comprehension of large software systems [1, 2, 9, 10]. In this work we use dynamic analysis and documented use-cases to extract execution traces of the software undergoing analysis. These traces are then used to map the use-cases to the source code that implements them. However, because we are interested in studying very large and intricate software, the mapping, albeit useful, should be analyzed further to produce abstractions that are cognitively tractable to software maintainers.

Our goal is to develop tools to assist software maintainers on perfective and adaptive maintenance tasks, which account for over two thirds of software maintenance efforts [4, 8]. To perform a maintenance activity, a developer's initial task is to study and analyze the source code and its documentation. For example, the task may be to modify the 'print' feature of the Mozilla web browser. The developer studies the source code to locate the portions that are related to the 'print' feature. For many software systems such as Mozilla, this task is difficult and time consuming, since the implementation of a feature may involve many classes and modules. In this context, modules refer to compiled binary objects mainly dynamically linked libraries (*i.e.*, binary DLLs). Moreover, many of the software relationships cannot be identified easily from the source code if the system implementation uses dynamically linked libraries and/or component-based models (*e.g.,* XPCOM, Mozilla's own COM-like component model).

In a typical system, more than half of the developer's effort is spent on reading and analyzing the source code to understand the system's logic and behavior [3, 4]. A complementary approach to code reading is to perform maintenance by instrumenting the source code, exercising the pertinent features using a profiler, and then analyzing the execution traces to determine the portions of the code that were exercised by the features.

This dynamic approach is suitable in practice because 'change requests' are usually written in natural language with explicit references to software features. A developer can start from the 'change request', then execute the application in a profiling mode, and finally exercise the desired features to locate the portions of the source code, instead of starting with the code and trying to map it to features manually.

This paper describes an approach to aid in the comprehension of large and intricate software such as the Mozilla web browser. Our approach is based on using profiling tools to map software features to source code. Specifically, our approach employs the concept of marked execution traces to define program features. A *feature* is defined as a use-case scenario such as `open-url` and `send-page`. Features are specified by the maintainer in terms of marked-traces. A *marked-trace* is established manually during the execution of the program by specifying the start and the end of the trace using a trace-marker utility that is part of the profiler.

After the software's features are specified, our tools analyze the traces to produce a set of views, at various levels of

	Numbers	Sizes (MLOC)
Header files	7,592	1.43
C files	1,980	1.09
C++ files	4,028	1.88
IDL files	1,998	0.18
C++, C & header files	13,600	4.41

Table 1. Mozilla files

detail, to assist the software maintainer in the comprehension of large software systems. Describing the tool is out of scope of this paper, for further details see [5, 6]. In this context we will show how the tool can be used to measure the intricacy of the software and the similarity between use cases.

2 The Mozilla case study

Mozilla is an open-source web browser ported to almost every operating system and hardware platform, also including tools such as an e-mail client, newsgroup reader, IRC (Internet Relay Chat), and an HTML editor. Mozilla's size ranges in the millions of lines of code (MLOC). It is developed mostly in C++. C code accounts for only a small fraction of the program. The Mozilla version we analyzed (Version 1.0.1) includes more than 13,000 source files for a total of up to 4.4 MLOC located in about 1,200 subdirectories. Mozilla also has over 3,000 support files with 1.1 MLOC of XML, HTML, perl and Javascript. Mozilla consists of over a 100 binary modules (DLLs) in addition to several executable objects such as `mozilla.exe`, which is the main executable, and installation programs.

The Mozilla use-cases mainly focus on the web browser and partially on the e-mail features. Table 1 outlines Mozilla's size. Clearly, the thousands of classes and relationships make program understanding and maintenance difficult.

As stated earlier, dynamic analysis with partial feature coverage[1] uncovered 119,571 unique invocation relationships between class methods, while the source code analysis performed, using SourceNavigator [7] on the entire source code distribution of Mozilla, found only 77,224 relationships between class methods. The source code distribution includes the source code of every Mozilla module and tool. The additional 42,347 relationships discovered via dynamic analysis were interactions between classes in various binary modules. A specific example is the `nsObserverService` class, which implements the `nsIObserverService` interface. The dynamic analysis uncovered 22 distinct relationships between `nsObserverService` and other classes in 11 binary modules. These relationships would not have been discovered using static analysis.

[1] Dynamic analysis is not able to fully exercise all features.

	Methods	Classes	Modules	XPCOM
Executed	30789			
Loaded	47892	4614	61	694
Total	77842	7875	102	1089

(a) Code coverage

Use-case	Modules	Classes	Methods	Events
save-page	42	2,950	8,684	17,139,817
print-page	39	2,848	8,739	15,074,421
open-url	43	3,004	9,432	33,192,788
bookmark-add	28	1,637	4,122	4,676,802
startup	46	3,342	9,456	49,739,968
open-link	42	2,667	8,311	14,349,139
bookmark-open	42	2,798	8,511	14,259,039
shutdown	58	2,479	5,011	11,473,102
send-page	54	3,792	12,301	71,743,527
unmarked traces	48	2,803	8,351	25,791,982

(b) Run-time statistics

Table 2. Use-case coverage statistics

2.1 The Mozilla use-cases

We identified an initial set of Mozilla use-cases that are characteristic of any web browser's functionality. Table 2 reports summary statistics as recovered by the dynamic analysis. In the tables, modules correspond to dynamically linked libraries and the main executable file of Mozilla `mozilla.exe`. Table 2(a) summarizes the overall runtime coverage of the use-cases. *Executed* counts the number of methods exercised, and *Loaded* counts the number of methods, classes, and modules loaded at runtime. A method is considered loaded when its container class is loaded, and a class is considered loaded when its container module is loaded. *Total* counts the total methods, classes, and modules in the binary code distribution of Mozilla. This total was extracted from the compiled binaries rather than the source code. Table 2(b) outlines the nine use-cases of the case study and their coverage statistics: number of modules, number of classes, number of methods, and the number of method-entry events created during the execution each use-case.

Next, we described the similarity between the use-cases. The use-case similarity matrix is computed from the caller-callee relationships of the methods invoked while executing each use-case. The similarity measure helps the engineer identify similar use-cases and, thus, guide him/her to learn about the implementation of a feature, or a use-case, by studying similar features. The similarity measure also helps the engineer to assess the impact of a change of one feature to the other features in the software system. In our case study, the similarities between some use-cases are ob-

	print-page	open-url	bookmark-add	open-link	startup	bookmark-open	shutdown	send-page
save-page	69	57	48	57	57	61	44	61
print-page	100	56	48	59	51	65	44	52
open-url		100	42	77	50	76	43	56
bookmark-add			100	43	40	48	42	36
open-link				100	45	84	46	49
startup					100	50	39	57
bookmark-open						100	47	53
shutdown							100	44
send-page								100

(a) Similarity of use-cases with all the modules and classes

	print-page	open-url	bookmark-add	open-link	startup	bookmark-open	shutdown	send-page
save-page	67	45	12	53	50	57	17	22
print-page	100	35	15	45	48	49	16	16
open-url		100	10	63	37	77	17	29
bookmark-add			100	10	10	11	9	3
open-link				100	36	81	16	20
startup					100	39	13	20
bookmark-open						100	20	21
shutdown							100	17
send-page								100

(b) Similarity of use-cases without the common modules

Table 3. Use case similarity matrices

vious (e.g., the strong similarity between the *open-url* and *bookmark-open* use-cases). The use-case similarity matrix, shown in Table 3, is computed using the Jaccard index similarity function, which is defined as:

$$Similarity(U_1, U_2) = \frac{|U_1 \cap U_2|}{|U_1 \cup U_2|}$$

where U_k is the set of caller-callee relationships of the method invoked while executing use-case k, $|U_1 \cap U_2|$ is the cardinality of the intersection of U_1 and U_2, and $|U_1 \cup U_2|$ is the cardinality of the union of U_1 and U_2.

During the analysis, common classes and modules can be included or filtered out. A common class or module is an entity that is used in the implementation of a high percentage of use-cases. Filtering out such entities not only reduces the clutter of the views, but also emphasizes the uniqueness of each use-case. Filtering out common modules also provides a better measure of similarity, for example, the similarity between semantically similar use-cases such as *open-link* and *bookmark-open*, does not change significantly if common classes and modules are filtered out. However, the similarity between *bookmark-add* or *send-page* and all other use-cases significantly decreases when the common classes and modules are filtered out.

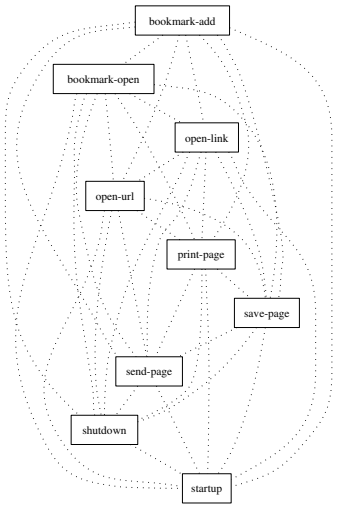

Figure 1. Use-cases view of Mozilla

2.2 Structural views

As stated in the introduction, the proposed approach and tool allow to extract some structural views from the runtime data. These views capture portions of Mozilla's architecture that pertain to each use-case. In this paper we will focus on the interaction view.

Figure 1 shows the use-case graph, the graph is almost a complete graph. This graph is a starting point to explore further details about each use-case. Each node in the graph encodes the module-interaction view of the modules that implement a use-case, while the edges encode the participating modules between two use-cases. The tool also allows to access module-interaction view for a given use-case can be viewed simply by selecting the node representing the use-case. The module-interaction view for a given use-case can be viewed simply by selecting the node representing the use-case. For example, an engineer can explore the *send-page* use-case by double-clicking on the *send-page* node, which will construct the module-interaction view of the *send-page* use-case as shown in Figure 2. In this view, nodes represent clusters of modules and edges represent the interaction between the modules in the clusters. The label of each cluster indicates the dominant module within the cluster. The module-interaction views can be annotated with simple metrics for each module that are helpful to assess the intricacy and the degree of interaction between other modules. An example of such metrics is the number of nodes and edges in the call graph of the module to highlight the

size of the call graph as shown in Figure 2. In the annotation **G** : N/E, N is number of classes in the module, and E is the number of relationships in the call graph of the module.

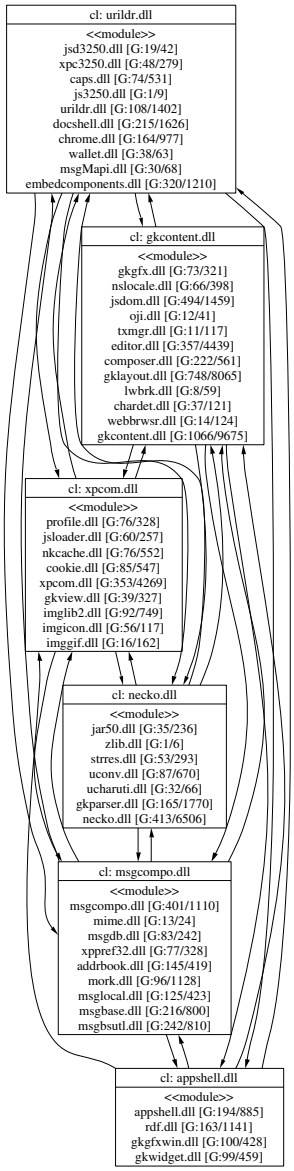

Figure 2. Module-interaction view (clustered) for the send-page use-case

3 Conclusions

In this paper we describe some preliminary results related to the comprehension of a large software system, Mozilla. The comprehension task was driven by use cases and dynamic analysis.

This allowed to and several views created using dynamic analysis that was driven by use-cases. Some views are based on a hierarchy of graphs that support the exploration of the system's software architecture. Other views are based on metrics and focuses on revealing the intricacy of the system.

Through the case study, we demonstrate the ability of our tools to collect dynamic data, analyze the data, and present it as a set of views. We believe that the software views and the automated tools described in this paper are helpful for maintenance tasks that require a detailed understanding of specific parts of a large software system.

References

[1] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *International Conference on Software Maintenance*, Florence, Italy, November 2001. IEEE.

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

[3] M. R. Olsem. Reengineering technology report. Technical Report Volume 1, Software Technology Support Center (STSC), October 1995.

[4] T. M. Pigoski. *Practical Software Maintenance: Best Practices Managing Your Software Investment*. John Wiley & Sons, 1997.

[5] M. Salah and S. Mancoridis. Toward an environment for comprehending distributed systems. In *Proceedings of Tenth Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, November 2003. IEEE.

[6] T. Souder, S. Mancoridis, and M. Salah. Form: A framework for creating views of program executions. In *International Conference on Software Maintenance*, Florence, Italy, November 2001.

[7] Source-Navigator IDE. http://sourcenav.sourceforge.net.

[8] B. Swanson and C. M. Beath. Departmentalization in software development and maintenance. *Communication of the ACM*, 33(6), June 1990.

[9] N. Wilde and M. Scully. Software reconnaissance: Mapping program features to code. *journal of Software Maintenance: Research and Practice*, 7(1), January 1995.

[10] E. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In *Proceedings of Application Specific Software Engineering and Technology (ASSET 99)*, Dallas, TX, March 1999.