Scenario–Driven Dynamic Analysis for Comprehending Large Software Systems

Maher Salah[1], Spiros Mancoridis[1], Giuliano Antoniol[2], Massimiliano Di Penta[3]
[1] Department of Computer Science, Drexel University
3141 Chestnut Street, Philadelphia, PA, 19104, USA

[2] Département de Génie Informatique, École Polytechnique de Montréal
P.O. Box 6079, Succ. Centre-Ville, Montréal (Québec), H3C 3A7, Canada

[3] RCOST - Department of Engineering, University of Sannio,
Palazzo ex Poste,Via Traiano, 82100 Benevento, Italy

{msalah,spiros}@cs.drexel.edu, antoniol@ieee.org, dipenta@unisannio.it

Abstract

Understanding large software systems is simplified when a combination of techniques for static and dynamic analysis is employed. Effective dynamic analysis requires that execution traces be generated by executing scenarios that are representative of the system's typical usage.

This paper presents an approach that uses dynamic analysis to extract views of a software system at different levels, namely (1) use cases views, (2) module interaction views, and (3) class interaction views. The proposed views can be used to help maintainers locate features to be changed.

The proposed approach is evaluated against a large software system, the Mozilla web browser.

1 Introduction

As reported in the literature [22] up to 90% of the software development cost is spent on maintenance and evolution activities. When performing a maintenance task, it is necessary to identify and to locate the portion of code that needs to be changed based on the intuition that is gained by executing the software and consulting its use-cases.

Locating a portion of code that needs to be changed is relatively easy when (1) the system is documented properly and (2) it is possible to trace high–level documents to the source code. However, this is not often the case because the only information about the system that is available is the source code itself. In such a case, the maintainers need to read and analyze the source code, which may require a significant amount of their available time [17, 18, 25]. For example, let us suppose that the task is to modify the 'print' feature of the Mozilla web browser. The developer studies the source code to locate the portions of the code that are related to the 'print' feature. For many large software systems this task is difficult and time consuming, since the implementation of a feature may involve many classes and modules.

To reduce such a labor-intensive effort, several approaches have been developed to identify the software's features automatically and to trace these features to the source code that implements them [6, 7, 29, 30]. Static analysis techniques are insufficient to understand a large software system and to identify its features, hence, static analysis techniques should be complemented by dynamic analysis techniques. This is often done by instrumenting the software's code, exercising its pertinent features using a profiler, and then analyzing the execution traces to determine the portions of the code that were exercised by the features. With adequate tool support, this approach enables developers to locate the code of interest quickly. It can also reveal run-time relationships between classes and between modules, as well as thread interactions for multi-threaded systems, which cannot be detected by static analysis alone.

This paper proposes an approach based on dynamic analysis to map use-cases to portions of the source code that implements them. This is done by exercising the system with scenarios obtained from use-case descriptions and change requests, and then collecting and analyzing execution traces. Since 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. 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 of the profiler. For example, immediately before the execution of the `open-url` use-case, the maintainer would press a button on the profiler's GUI indicating the beginning of the marked trace. After the URL opens, the maintainer would press another button of the GUI to signify the end of that feature (or marked trace).

After the software's features are exercised, our tools analyze the traces to produce a set of views, at various levels of detail, to assist the software maintainer in the comprehension of large software systems. The views are created by subjecting the software systems to dynamic analysis under various use-case scenarios. Two sets of views are constructed 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 use-cases.

This dynamic approach works well in practice, since '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.

A major challenge of dynamically analyzing large systems is the significant degradation in performance and the lack of profilers that are suitable for software comprehension. For example, many of the dynamic profilers focus on analyzing the performance of Win32-based software; other tools are based on the concept of statistical profiling, thus infrequent events can be lost. Furthermore, profiling becomes even more challenging when profiling multi-threaded and multi-language software systems.

The organization of the rest of the paper is as follows: Section 2 presents related research on feature identification and dynamic analysis. Section 3 describes the tools used to support our approach. Section 4 shows how the proposed approach is applied to a large software system, namely the Mozilla browser. Section 5 concludes and outlines directions for future work.

2 Related work

This work is related to two research areas, namely dynamic analysis and program feature analysis. The next two subsections briefly summarize the types of dynamic information applicable for analyzing software systems and the different approaches for program feature analysis.

2.1 Dynamic analysis

Dynamic analysis is used to study the behavior of software systems. Even if approaches based solely on static information exist, hybrid approaches integrating static and dynamic information provide a more accurate or a faster solution.

There are three methods for collecting runtime data. The first method is source code instrumentation. Bruegge *et al.* designed the BEE++ system [3] as a framework for monitoring (*e.g.,* function calls, variable modifications) systems written in C/C++. In this system, runtime event generation is achieved by instrumenting the program source code. Another system that uses source code instrumentation is SCED [12]. This system uses runtime data to create models of object-oriented programs, which are visualized as state diagrams or state charts. SCED only collects data from standalone applications, while BEE++ can also collect data from distributed applications.

The second method for collecting runtime data involves the instrumentation of compiled code. This method is widely used to instrument Java bytecode [13]. The third runtime data collection method is based on debugging and profiling. In this method, code instrumentation is not required. Modern development frameworks provide interfaces to facilitate the collection of runtime data. Examples of such interfaces include JVMDI and JVMPI for Java [26, 27], CLR Profiling Services for Microsoft .NET [16], and COM+ instrumentation services for COM+ applications [15]. Debuggers have been used to emulate profiling interfaces by automatically inserting breakpoints and manipulating the stack frame of the executing program. For example, GDBProfiler [5] uses the GNU debugger interface to profile C programs.

2.2 Program feature analysis

The objective of feature analysis is to correlate program features with implementation artifacts found in the source code. In this context, a feature usually refers to a usage scenario of the program [6, 7] or a test case [30].

The most closely related research work is by Eisenbarth *et al.*, who use a combination of dynamic and static analysis to associate features to components [6] in two medium sized web browsers (i.e., Mosaic and Chimera). Dynamic profiling is used to identify the subprograms that are exercised when a feature is executed. Our work differs from Eisenbarth's work in three ways. First, in our approach, program features are identified while the user interacts with the program. Hence, there is no restriction on the order of the execution of features, because the user can determine the start and the end of each feature interactively with the aid of the trace-marker utility embedded in the profiler. Eisen-

barth's approach requires starting and ending the program for every scenario. Second, the focus of this work is on the analysis of large systems, the size of the program in our case study, Mozilla, is in over four million lines or code, while the Mosaic and Chimera sizes are in the range of fifty thousand lines of code. Third, the analysis and software views produced by our tools are different. Their tools produce concept analysis views, which differ significantly from the views produced by our analysis.

Wong *et al.* used program execution slices to identity the portions of the code that implement a given feature or a set of features [30]. In their precursor work, Wilde and Scully [29] propose a technique to identify features by analyzing execution traces of test cases. They use two sets of test cases to build two execution traces: an execution trace where a functionality is exercised; and an execution trace where the functionality is not. Then, they compare execution traces to identify the feature associated with the functionality in the program. In their work, the authors only use dynamic data to identify features, no static analysis of the program is performed. Chen and Rajlich [4] develop a technique to identify features using Abstract System Dependencies Graphs (ASDG). In C, an ASDG models functions and global variables as well as function calls and data flow in a program source code. Maintainers identify, manually, features using the ASDG following a precise process. Unlike Wilde and Scully's work, Chen and Rajlich use only static data to identify features and a manual search, which is prohibitive for large multi-threaded object-oriented programs.

The evolution of OO software entities from a features point of view is the topic of Greevy *et al.* [10]. Multiple versions of a system are analyzed to obtain dynamic information on the evolution of a class' role. Dynamic and static information is also exploited by Antoniol *et al.* [2]. Probabilistic ranking and knowledge base filtering are used to filter information extracted by executing feature dependent and feature independent scenarios. The contribution of methods and classes to a given feature are highlighted as elements of a micro-architecture that implements the feature of interest. Finally, Eisenberg and De Volder [8] propose to partition test cases into feature-specific subsets.

Our recent work [20] relies on both static and dynamic data to identify features in Java programs. It goes beyond feature identification by creating feature-interaction views, which highlight dependencies among features. Recently, feature identification and evolution has been the subject of several new studies. In particular, our tools [20] were applied to Mozilla and several new views were proposed [21]. Our recent paper [21] showed early results of the present work, presenting statistics from the Mozilla dynamic analysis and introducing the idea of extracting structural views from runtime data.

3 Software comprehension environment

This section describes the software comprehension environment that has been used to create the software views described in Section 4. Further details about the environment are described elsewhere [19, 23]. Figure 1 illustrates the architecture of the software comprehension environment. The main subsystems are: data gathering, repository, and analysis/visualization.

The **data gathering subsystem** defines the interfaces and data formats of the data collection tools (*i.e.,* static analyzers and dynamic profilers). In our Mozilla case study, profiling is performed using code instrumentation by inserting function handlers to intercept function-entry and exit events. The function-entry handler was implemented using the Microsoft C++ compiler's `/Gh` option, while the function-exit handler was implemented by manipulating the call stack and forcing the program to invoke our own function-exit handler. Function entry and exit events are passed to the **data collection module**, which marks each event with the current marked-trace label (chosen by the user). The data collection module builds the function call-graph of the program and maintains function call statistics such as the total execution time, number of times a method is invoked, and the number of exceptions thrown by each method. Upon the termination of the program's execution, the collected data is stored in flat files, which are subsequently imported into the repository for analysis.

The **repository subsystem** defines the data and meta-data models, as well as the data manipulation and query language. The data repository stores the program entities, relationships, and runtime events that are gathered from the subject system. The repository is manipulated using standard SQL and is queried using either SQL or our own SMQL query language [19]. The repository uses any JDBC-compliant database.

SMQL (Software Modelling Query Language) simplifies the data retrieval and analysis of program data to create software views. Even though the repository can be queried using SQL, designing queries for comprehending software systems using SQL is cumbersome. Many of the queries that are of interest to an engineer, for example queries that involve the transitive closure of a relation, are not supported directly by SQL. SMQL is a set-based language that facilitates the definition of queries about entities, relations, and events by translating the SMQL code into SQL query statements. SMQL provides a built-in `closure` function as well as binary operators such as union, intersection, and join. SMQL is similar to `grok` [11] for manipulating binary relational algebra expressions. Unlike `grok`, SMQL can be extended to support additional operations that are implemented in Java such the computation of runtime interactions between program features [20].

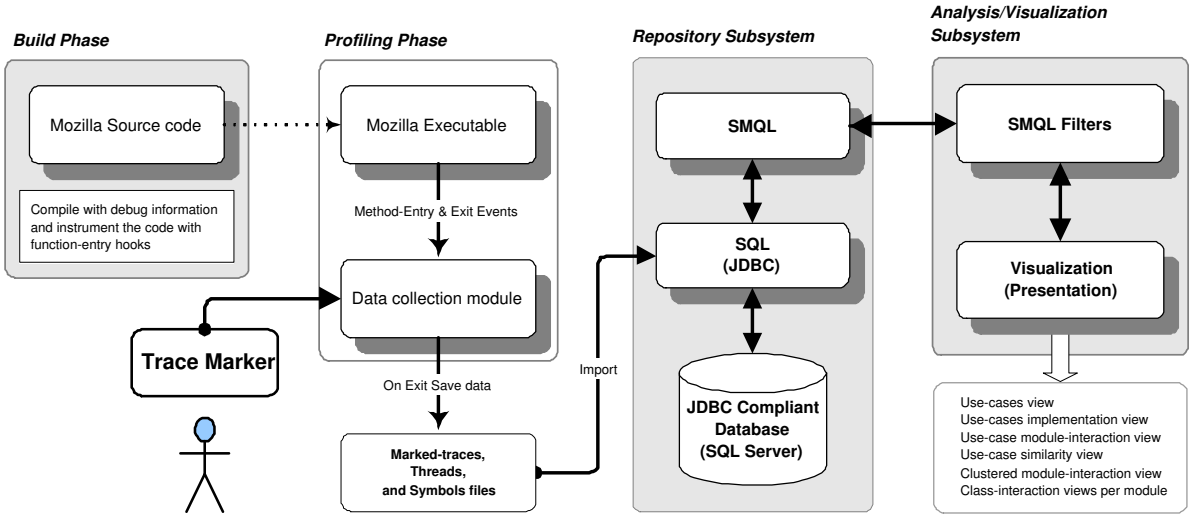


Figure 1. The Software Comprehension Environment

The **analysis and visualization subsystem** is responsible for the creation and visualization of software views. Analysis is performed using SMQL analyzers, which are SMQL extensions that are implemented in Java. Software views are visualized as graphs. The graph visualization is performed using JGraph [1], graph layout is performed by dot [9], and graph clustering is performed by Bunch [14]. Software views are implemented as SMQL extensions written in Java. The tool supports the navigation among the different views. Depending on the graph of a view, a node can be a feature, a module, a class, or a method.

The views are constructed as a hierarchy with different levels of abstraction (Figure 2). The lowest level represents the caller-callee relationships between methods that are invoked during the execution of the use-cases. The second level of abstraction includes class-to-class relationships. The third level includes module-to-module relationships. The highest level of the hierarchy represents program uses-cases. As stated earlier, a use-case refers to an externally visible functionality of the program, and it is identified by the developer as a marked-trace.

View exploration is performed by selecting one or more nodes in the graph. Selecting a single node will produce the lower-level view of the selected entity. For example, selecting a use-case node produces the module-interaction view of the use-case. While, selecting more than one node produces a view that contains the selected nodes and the detailed interactions between them. For example selecting two use-cases nodes produces a view with the two use-cases and the modules shared between the use-cases as nodes, and the *uses* relationships between the modules as edges.

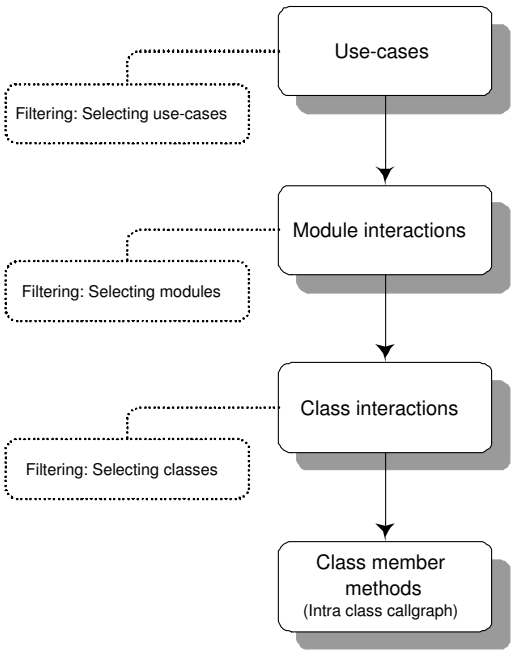


Figure 2. Hierarchical view navigation

	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

	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

4 Case study: tracing features of the Mozilla web browser

This section describes a case study related to the understanding of the Mozilla web browser. Mozilla is an open-source web browser ported to almost every operating system and hardware platform. In addition to the web browser as the core functionality, Mozilla includes other Internet tools such an e-mail client, newsgroup reader, IRC (Internet Relay Chat), and an HTML editor.

Mozilla's size ranges in millions of lines of code (MLOC). It is developed mostly in C++. The Mozilla version we analyzed (Version 1.0.1) includes more than 13,000 source files (see Table 1), 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 100 binary modules (DLLs) in addition to several executable objects such as `mozilla.exe`, which is the main executable, and several installation programs.

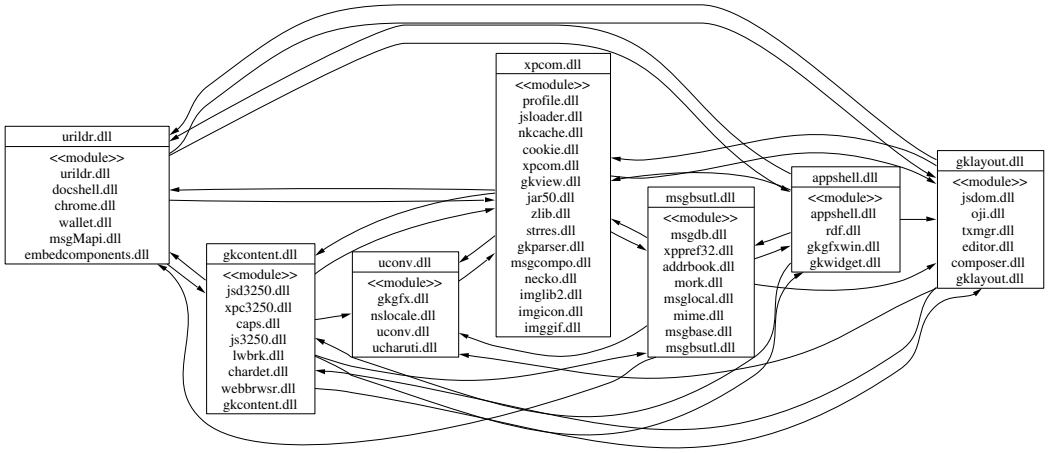
The use-cases reported in the present case study mainly focus on the web browser and partially on the e-mail features of Mozilla. Clearly, the thousands of classes and relationships complicate program understanding and maintenance.

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 of each use-case.

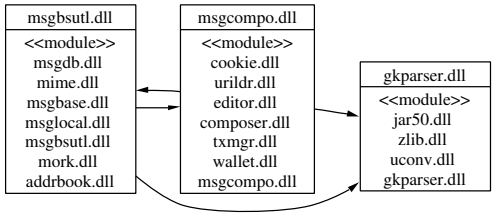
In our analysis, dynamic analysis with partial feature coverage uncovered 119,571 unique invocation relationships between class methods, while the source code analysis performed, using SourceNavigator [24] on the entire source code distribution of Mozilla, found only 77,224 relationships between class 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.

4.1 Scenario-driven dynamic analysis

To illustrate how use-case driven dynamic analysis can help maintainers, let us suppose that the maintenance task at hand is to modify the *send-page* feature of the Mozilla browser. The *send-page* feature allows a user to send an HTML page or an HTML link via e-mail. The maintainer can study the source code and the available documentation of the system, and try to locate the portions of the source code related to the *send-page* use-case. Alternatively, the maintainer can use our approach to exercise the *send-page* use-case and locate the portions of the source code. To simplify the software views produced by the analysis, it is recommended that the maintainer exercises other features of the program in addition to the feature of interest. The reason for exercising other features is to be able to determine the common modules and classes exercised by most features.



(a) Common modules/classes are included



(b) Common modules/classes are filtered out

Figure 3. Module-interaction view (clustered) for the *send-page* use-case

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.

For large systems like Mozilla, the graphs representing the views tend to be large with hundreds of relationships, which makes some of the graphs unmanageable. To simplify these graphs, two steps have been taken: (1) provide the option of whether to include or exclude the commonly shared classes and modules in the analysis (this step was useful, because, on average, 50% of the classes referenced are part of the common infrastructure of the program); and (2) cluster the graphs of each view. The clustering tool (Bunch [14]) uses optimization to maximize cohesion between nodes in the same cluster and minimize coupling between nodes in different clusters. For example, without filtering the common modules, the resulting module-interaction (clustered) view for *send-page* is shown in Figure 3(a). A simplified module-interaction (clustered) view for *send-page* is obtained by filtering out the common modules is shown in Figure 3(b). In both views shown in Figure 3, nodes represent clusters of modules and edges represent the interactions between the modules in the clusters. The label of each cluster indicates the dominant module within the cluster. The dominant module in a cluster has the highest number of fan-in and fan-out edges.

Next, we describe the similarity matrix between the use-cases and some of the structural views using the *send-page* feature as an example.

4.2 Use-case similarity matrix

We next analyze the similarity between Mozilla 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 to identify similar use-cases and, thus, guides him to

	print-page	open-url	bookmark-add	open-link	startup	bookmark-open	shutdown	send-page
save-page	69 (67)	57 (45)	48 (12)	57 (53)	57 (50)	61 (57)	44 (17)	61 (22)
print-page	100 (100)	56 (35)	48 (15)	59 (45)	51 (48)	65 (49)	44 (16)	52 (16)
open-url		100 (100)	42 (10)	77 (63)	50 (37)	76 (76)	43 (17)	56 (29)
bookmark-add			100 (100)	43 (10)	40 (10)	48 (11)	42 (9)	36 (3)
open-link				100 (100)	45 (36)	84 (81)	46 (16)	49 (20)
startup					100 (100)	50 (39)	39 (13)	57 (20)
bookmark-open						100 (100)	47 (20)	53 (21)
shutdown							100 (100)	44 (17)
send-page								100 (100)

(The numbers in the table are percentages)

Table 3. Use case similarity matrices (in parentheses without considering common modules)

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 on the other features in the software system. In our case study, the similarities between some use-cases are obvious. For example, the strong similarity between the *open-url* and *bookmark-open* use-cases.

The similarity between two use-cases is measured by approximately matching the graphs that represent the call-graphs of two use-cases U_1 and U_2 with associated call-graphs $G_1(V_1, E_1)$ and $G_2(V_1, E_2)$ respectively. The similarity between U_1 and U_2 is computed as:

$$Similarity(U_1, U_2) \quad = \quad \frac{|E_1 \cap E_2|}{|E_1 \cup E_2|}$$

where V_k is the set of nodes in G_k, which is the set of methods invoked during the execution of use-case U_k. E_k is the set of edges in G_k, which is the set of call relationships in the call-graph of feature U_k. Each element of E_k represents the call relationship between two methods. Edges are matched by comparing the tail and the head node of the edge. This similarity measure is based on set similarity measure, which is referred to as the Jaccard Index. For our purpose, this measure is an acceptable approximation for matching call-graphs [28].

As stated earlier, 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 (see values between parentheses in Table 3), 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.

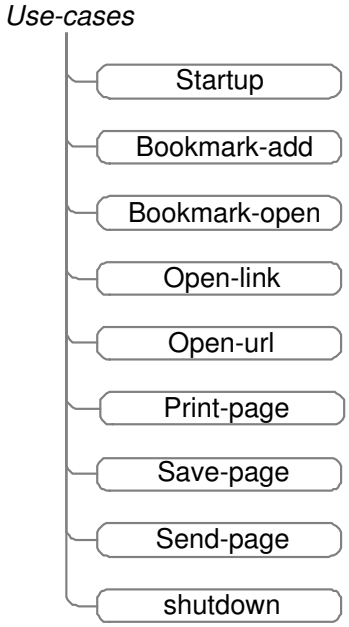


Figure 4. Use-cases view of Mozilla

4.3 Structural views

As described in Section 3 and shown in Figure 2, the understanding of a large software system starts by browsing the use-cases view. Figure 4 shows the Mozilla use-cases. 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. The module-interaction view for a given use-case can be viewed simply by selecting the node representing the use-case.

Let us suppose now that a maintainer needs to modify the *send-page* feature (i.e., the feature that allows a user to send

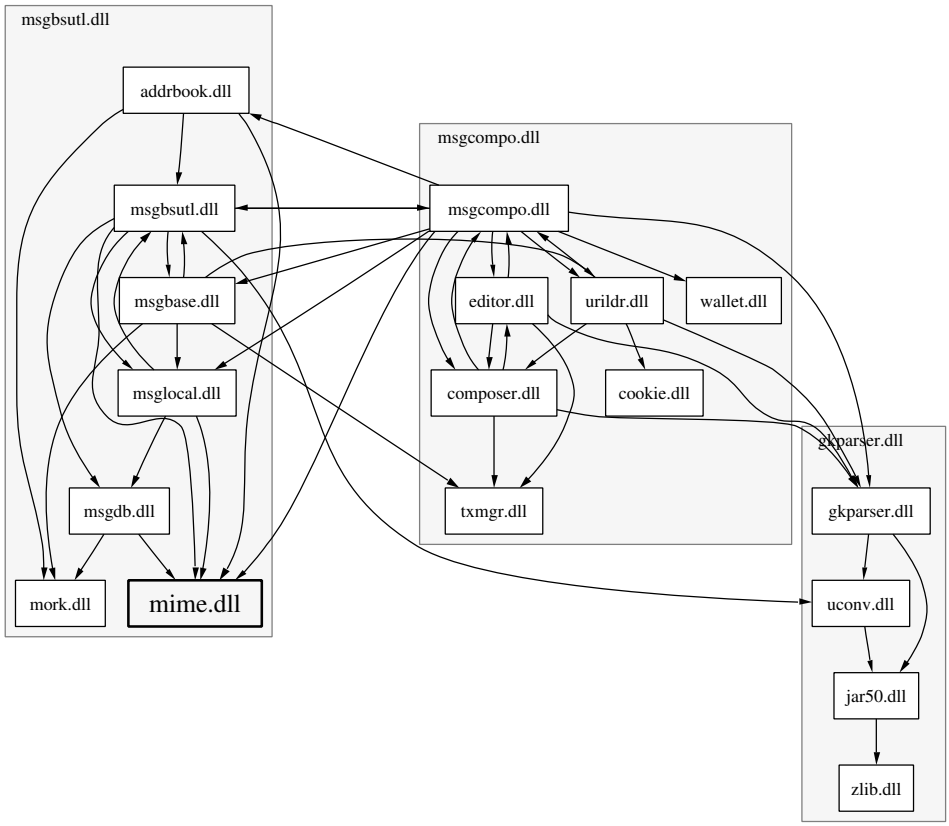


Figure 5. *Send-page* **module-interaction view (clusters are expanded and common modules/classes are filtered out)**

an HTML page via e-mail). The maintainer 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 5. In this view, the nodes in each cluster are expanded to show the interactions between modules. The modules related to the *send-page* feature are those contained within the `msgbsutil.dll` cluster. These modules are only used (with the exception of *shutdown* feature) by the *send-page* use-case.

Further exploration can be performed at the module level. Let us suppose that, for the *send-page* feature, the maintainer wants to focus on the module dealing with *MIME* types for encoding the messages. The user clicks on the `mime.dll` module to open its class-interaction view shown Figure 6. In Figure 6, nodes represent classes and edges represent invocations between the member methods of the classes.

A complementary view of the class interaction view is the module-dependency view centered around the module of interest (`mime.dll` in this case), which highlights the modules that the `mime.dll` module directly interacts with, as shown in Figure 7. This view is constructed by selecting the module of interest and specifying to include modules that it interacts with directly (its direct neighbors). In this graph, nodes represent modules, and edges represent the invocations between the classes of the modules.

5 Conclusions

This paper describes a case study of a large software system, Mozilla, 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 focus 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. 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.

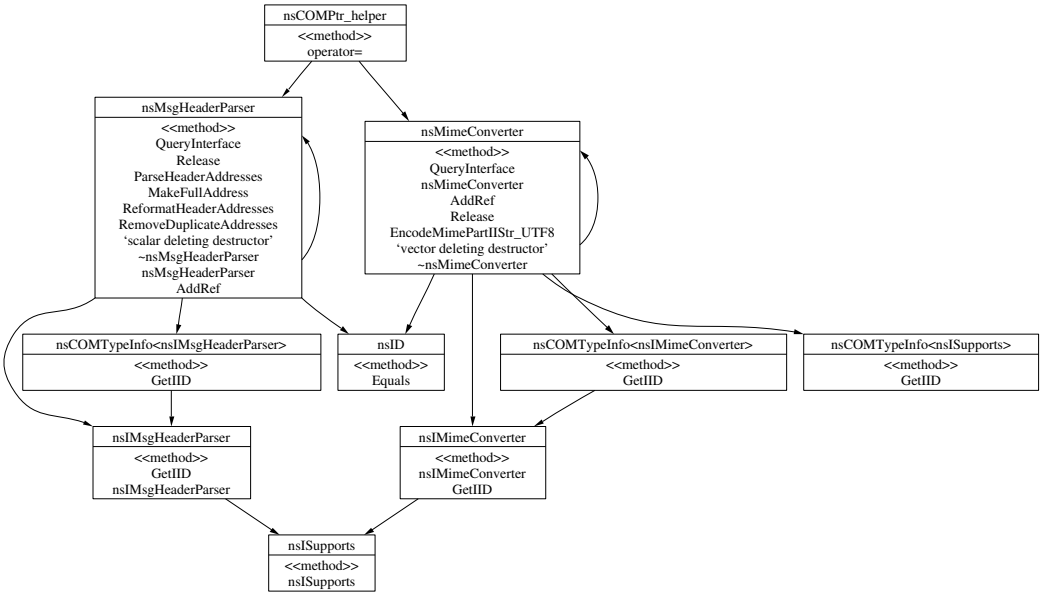


Figure 6. Class-interaction view for the `mime.dll` module

It is worth noting that the approach described in this paper has some limitations. First, is the degradation in performance when profiling large systems. Second, at the time of writing this paper, our approach can only be applied to programs whose components execute in the same process space. For example, programs that use Microsoft COM+ [15] components cannot be profiled completely, because the profiler cannot collect events from COM+ components. COM+ components require a separate profiler to collect runtime events from the execution of the components using the COM+ instrumentation services. Third, building a complete view of the runtime behavior of a software system is a difficult task, because of the inherent difficulty of executing all possible use-cases of a program. Our approach is best used to execute a subset of features (*as-needed*) that are relevant to a specific maintenance task.

References

[1] G. Alder. *Design and Implementation of the JGraph Swing Component*, 2003. http://www.jgraph.com.

[2] G. Antoniol and Y. Gueheneuc. Feature identification: a novel approach and a case study. In *International Conference on Software Maintenance*, pages 357–366, Budapest, Hungary, Sept 26-29 2005. IEEE Press.

[3] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analysis. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSLA93)*, Washington, USA, September 1993.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In A. von Mayrhauser and H. Gall, editors, *proceedings of the 8^{th} International Workshop on Program Comprehension*, pages 241–252. IEEE Computer Society Press, June 2000.

[5] C. Dahn and J. Penrose. GDBProfiler for GNU C/C++. http://serg.mcs.drexel.edu/gdbprofiler.

[6] 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.

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

[8] A. D. Eisenberg and K. De Volder. Dynamic featire traces: Finding features in unfamiliar code. In *International Conference on Software Maintenance*, pages 337–346, Budapest, Hungary, Sept 26-29 2005. IEEE Press.

[9] E. Gansner, E. Koutsofios, and S. C. North. *Drawing graphs with dot*. AT&T Bell Laboratories, Murray Hill, NJ, February 2002.

[10] O. Greevy, S. Ducasse, and T. Girba. Analyzing feature traces to incorporate the semantic of changes. In *International Conference on Software Maintenance*, pages 347–356, Budapest, Hungary, Sept 26-29 2005. IEEE Press.

[11] R. C. Holt. Binary relational algebra applied to software architecture. Technical Report CSRI-345, University of Toronto, March 1996.

[12] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A tool for dynamic modelling of object systems. Technical Report A-1996-4, Department of Computer and Information Sciences, University of Tampere, Finland, 1996.

[13] H. Lee and B. Zorn. *BIT: Bytecode Instrumenting Tool*. http://www.cs.colorado.edu/ hanlee/BIT.

[14] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *International Conference*

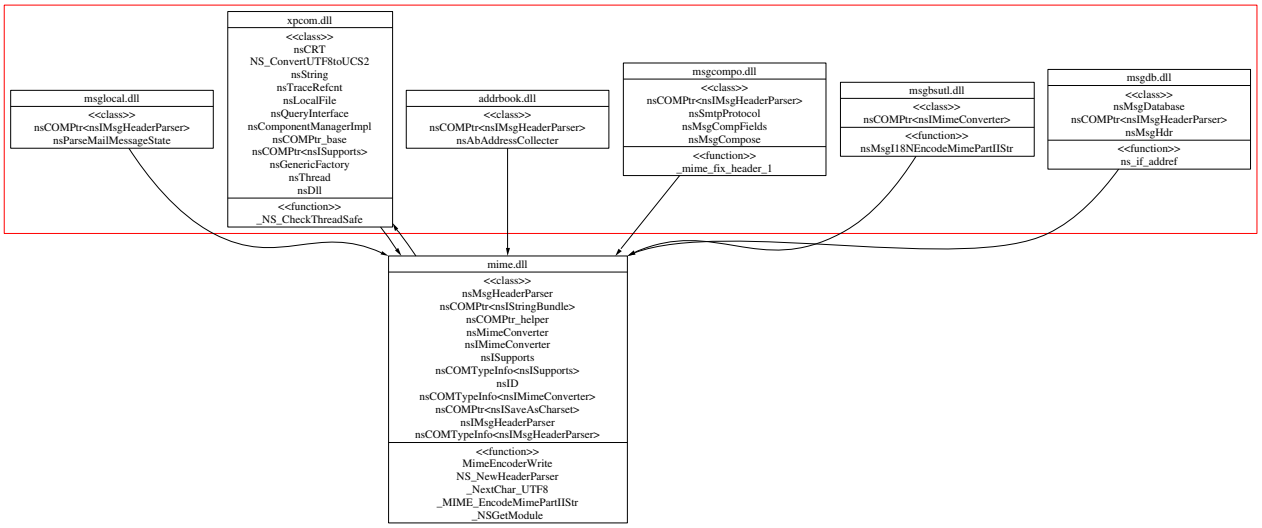


Figure 7. Module dependency view of `mime.dll` module

on Software Maintenance, pages 50–59. IEEE Computer Society Press, 1999.

[15] Microsoft Corporation. *COM+ SDK Documentation: COM+ Instrumentation*, 1999.

[16] Microsoft Corporation. *.NET Framework: Runtime profiling*, 2001.

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

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

[19] 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.

[20] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. In M. Harman and B. Korel, editors, *proceedings of the 20^{th} International Conference on Software Maintenance*, pages 72–81. IEEE Computer Society Press, September 2004.

[21] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Towards employing use-cases and dynamic analysis to comprehend mozilla. In *International Conference on Software Maintenance*, pages 639–642, Budapest, Hungary, Sept 26-29 2005. IEEE Press.

[22] R. Seacord, D. Plakosh, and G. Lewis. Modernizing legacy systems: Software technologies, engineering processes, and business practices. *SEI Series in Software Engineering*, 2004.

[23] 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.

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

[25] T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 5(10):494–497, 1984.

[26] Sun Microsystems, Inc. *Java Platform Debugger Architecture*, 1999.

[27] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI)*, 1999.

[28] W. D. Wallis, P. Shoubridge, M. Kraetz, and D. Ray. Graph distances using graph union. *Pattern Recognition Letters*, 22:701–704, May 2001.

[29] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. In K. H. Bennett and N. Chapin, editors, *Journal of Software Maintenance: Research and Practice*, pages 49–62. John Wiley & Sons, January-February 1995.

[30] 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.