An Environment for Comprehending the Behavior of Software Systems

A Thesis
Submitted to the Faculty
of
Drexel University
by
Maher M. Salah
in partial fulfillment of the
requirements for the degree
of
Doctor of Philosophy
July 2005
Dedication

*In memory of my Mother (1935-1992)*
Acknowledgements

I would like to express my deepest gratitude and appreciation to my advisor Professor Spiros Mancoridis, who inspired my interest in software engineering and software comprehension. Prof. Mancoridis has spent countless hours of discussions of all aspects of my research. His continuous support, encouragement and guidance are admirable. Also, I would like to thank the other members of my thesis committee: Dr. A. Shokoufandeh, Dr. E. Gansner, Dr. V. Prevelakis, Dr. F. Vokolos for their input and thoughtful feedback. I would like to thank two former committee members, Dr. Hewett and Dr. Souter, for their input and contribution to my work. I would like to thank and recognize Dr. Vokolos, Dr. Shokoufandeh and his graduate student T. Denton for their contribution and the countless discussion sessions I had with them.

I am grateful to the National Science Foundation and Lockheed Martin Corporation for sponsoring my equipment and travelling expenses to attend conferences.

I would like to express my gratitude and appreciation to the University of Pennsylvania Health System for their sponsorship and the financial support throughout my journey. Also, I would like to thank all members of the Web Applications Group. Specially, I want to recognize Mark Newman and Ying Xing for making work a little bit more fun. Also, I would like to recognize Tod Simons, Rich Urbani, Ginny Barry and Todd Kirkes.

Finally, I want to acknowledge my beloved family for their emotional support and continuous encouragement.
Table of Contents

List of Tables ........................................................................ vii

List Of Figures ...................................................................... viii

Abstract ................................................................................ xi

Chapter 1. Introduction .......................................................... 1
  1.1 Our Research ................................................................. 3
  1.2 Dissertation Outline ....................................................... 7

Chapter 2. Background .......................................................... 9
  2.1 Software Views ............................................................... 11
     2.1.1 Techniques for Graph-based Software Views ................. 13
  2.2 Analysis of Multi-language Applications ............................ 15
  2.3 Analysis of Distributed Systems ........................................ 16
  2.4 Analysis of Program Features ......................................... 17
  2.5 Summary ......................................................................... 18

Chapter 3. Software Comprehension Environment ..................... 20
  3.1 Data Gathering ............................................................... 21
     3.1.1 Remote Interaction ..................................................... 22
     3.1.2 Feature-driven Profiling ............................................. 25
  3.2 Data Repository ............................................................. 27
     3.2.1 Language Definition Model ....................................... 27
     3.2.2 Program Data Model ............................................... 30
  3.3 SMQL: Query Language ................................................ 32
     3.3.1 SMQL Objects ........................................................ 35
     3.3.2 SMQL Query .......................................................... 36
     3.3.3 SMQL Filters ........................................................ 40
  3.4 Analysis and Visualization .............................................. 43
  3.5 Summary ......................................................................... 46

Chapter 4. Dynamic Software Views ........................................ 52
  4.1 Call-graph View ............................................................. 54
  4.2 Object-interaction View .................................................. 55
  4.3 Class-interaction View .................................................... 57
  4.4 Module-interaction View ................................................ 62
  4.5 Thread-interaction View .................................................. 62
Chapter 5. Case Study .................................................. 78
  5.1 Mozilla: Web Browser ..................................... 82
    5.1.1 Mozilla: Build and Profiling ...................... 83
    5.1.2 Mozilla: Features .................................. 86
    5.1.3 Mozilla: Structural Views ......................... 88
  5.2 Jext: Text Editor ........................................... 96
    5.2.1 Jext: Build and Profiling ...................... 100
    5.2.2 Jext: Features .................................. 100
    5.2.3 Jext: Structural Views ......................... 106
    5.2.4 Jext: Protocol View .......................... 106
  5.3 TechReport: Publication Database ..................... 112
    5.3.1 TechReport: Build and Profiling .............. 113
    5.3.2 TechReport: Features ........................ 116
    5.3.3 TechReport: Structural and Remote-interaction Views 121
  5.4 Summary .................................................... 122

Chapter 6. Conclusions ............................................. 127

Appendix A. Software Modeling Query Language (SMQL) ......... 130

Appendix B. User Manual ........................................... 133
  B.1 Setup and Installation .................................. 133
  B.2 Data Collection Tools ................................... 135
    B.2.1 Wdbg: Profiler for Windows Applications .... 135
    B.2.2 WinProf: Win32 Profiler Library ................. 140
    B.2.3 JvProf: Java Profiler .......................... 141
    B.2.4 Jsa: Java Static Analyzer ....................... 144
    B.2.5 SymDbg: Win32 Static Analyzer ................. 144
  B.3 Distributed Profiler .................................... 145
  B.4 SDTool: Sequence Diagrams ............................. 147
  B.5 Running Scetool ........................................... 147
  B.6 Summary .................................................... 155

Bibliography .......................................................... 162

Vita ................................................................. 168
List of Tables

4.1 Sequence segment alignment .................................................... 75
5.1 Systems analyzed ................................................................. 78
5.2 Outline of software views produced for systems of the case study .......... 81
5.3 Mozilla files ................................................................. 82
5.4 Mozilla code coverage ............................................................ 83
5.5 Features coverage statistics ..................................................... 87
5.6 Feature similarity matrices ...................................................... 89
5.7 Jext: Coverage statistics .......................................................... 100
5.8 Jext: Feature coverage statistics ............................................. 102
5.9 Jext: Feature similarity matrix .................................................. 106
5.10 TechReport: Features coverage statistics ................................... 117
5.11 TechReport: Feature similarity matrix (Server-side) ....................... 120
5.12 TechReport: Feature similarity matrix (Client-side) ....................... 120
List of Figures

2.1 Process flow of constructing software views ........................................... 11
3.1 Architecture of the software comprehension environment .......................... 21
3.2 Data Gathering Subsystem ................................................................. 22
3.3 Communication Endpoints ...................................................................... 23
3.4 Endpoint entity and connect relation ....................................................... 24
3.5 Distributed profiling .............................................................................. 26
3.6 Java Language data model ...................................................................... 28
3.7 Language definition model ..................................................................... 29
3.8 Program data schema ............................................................................ 30
3.9 SMQL: Inheritance example ................................................................... 33
3.10 Inheritance example ............................................................................ 34
3.11 SMQL code example ............................................................................ 37
3.12 SMQL result tree ................................................................................ 38
3.13 Analysis and visualization subsystem .................................................... 43
3.14 Visualization process flow ..................................................................... 45
3.15 SMQL: Attaching a user-defined visualizer to an event set ...................... 46
3.16 SeeTool: SMQL result tree ................................................................... 47
3.17 SeeTool: Default visualizer for Relation-Set ....................................... 48
3.18 SeeTool: Default visualizer for Event-Set ............................................ 49
3.19 SeeTool: Sequence diagram visualizer for Event-Set .............................. 50
4.1 Hierarchy of views ................................................................................ 52
4.2 Static call graph for the sorting example ............................................... 55
4.3 Dynamic call-graph (QuickSort) ............................................................. 56
4.4 Dynamic call-graph (BubbleSort) .......................................................... 56
4.5 KWIC: Object-interaction view ............................................................... 58
4.6 KWIC: Class-interaction view (with objects shown) .............................. 59
4.7 KWIC: Class-interaction view (based on object creation and usage) ...... 60
4.8 KWIC: Class-interaction view (based on method invocation) ................. 61
4.9 Module-interaction view for imgjpeg.dll module .................................. 63
4.10 Thread-interaction view ...................................................................... 64
4.11 Remote-interaction view for the search-by-author feature of TechReport 65
4.12 Feature-interaction grid ....................................................................... 67
4.13 Feature-interaction and implementation views ..................................... 67
4.14 Analysis subsystem: Class-usage estimation ....................................... 73
5.1 Process flow of program analysis ............................................................ 78
5.2 dbimport tool .................................................. 79
5.3 SceTool: Environment main window ....................... 80
5.4 Tool Architecture for the analysis of Mozilla .......... 84
5.5 Mozilla: Hierarchical view navigation .................... 90
5.6 Mozilla features .............................................. 91
5.7 Module-interaction view (clustered) for the send-page feature ................................................. 92
5.8 Feature-implementation view ................................... 93
5.9 SMQL: Mozilla feature analysis .............................. 94
5.10 Class-interaction view for the mime.dll module ........ 95
5.11 Module interaction view of mime.dll module ............. 95
5.12 SMQL: Mozilla class/module interaction example ........ 97
5.13 SMQL (Rewrite): Mozilla class/module interaction example ....................................................... 98
5.14 Jext: Programmer’s text editor ............................. 99
5.15 Tool Architecture for the analysis of Jext ............... 101
5.16 Jext: SMQL code ............................................. 102
5.17 Jext: Feature-interaction view .............................. 103
5.18 Jext: Feature-implementation view ........................................... 104
5.19 Jext: Result window ........................................ 105
5.20 Jext: Hierarchical view navigation ......................... 107
5.21 Jext: Clustered class-interaction view of the open-document feature .............................................. 108
5.22 Jext: Class-interaction of org.jext.JextFrame cluster .................................................. 109
5.23 Jext: SMQL code for the open-document feature ........ 110
5.24 TechReport architecture .................................... 113
5.25 Enterprise Java Bean Architecture ......................... 114
5.26 Tool Architecture for the analysis of TechReport .... 116
5.27 TechReport: SMQL code .................................... 118
5.28 TechReport: Feature-interaction view ........................ 118
5.29 TechReport: Feature-implementation view ................ 119
5.30 TechReport: Hierarchical view navigation ................. 121
5.31 TechReport: Class-interaction view of the add-publication feature .............................................. 123
5.32 TechReport: Class-interaction view of the add-publication feature (Client-side) ................................. 124
5.33 TechReport: Class-interaction view of the search-by-title feature .............................................. 125
5.34 TechReport: Thread-interaction view ........................ 126

B.1 SCE: Directory structure ........................................ 134
B.2 Wdbg: Win32 profiler .......................................... 137
B.3 Wdbg: An example of the profiler configuration file .... 139
B.4 Jsa: Java static analyzer tool ................................ 145
B.5 Distributed profiler ........................................... 146
B.6 KWIC: Importing runtime data ............................... 148
B.7 KWIC: Running SceTool ....................................... 149
B.8 KWIC: SceTool after loading the SMQL code ............ 150
B.9 KWIC: SceTool after executing the SMQL code .......... 151
B.10 KWIC: Viewing software views .............................. 152
B.11 Java source code for IFilter interface .................... 153
B.12 Java source code for the AbstractFilter abstract class .................................................. 154
B.13 Importing GroupObjects class in the SMQL code ....... 156
B.14 SMQL results from executing the GroupObjects filter .................................................. 157
Abstract

An Environment for Comprehending the Behavior of Software Systems

Maher M. Salah
Dr. Spiros Mancoridis

Modern software systems are often large, distributed, written in more than one programming language, and developed using pre-built components (e.g., dynamically-linked libraries). The characteristics of such systems pose unique challenges to software comprehension, where the current practice typically involves the manual inspection and possibly static analysis of the system’s source code.

Our research addresses the characteristics of these systems by developing a software comprehension framework that facilitates dynamic analysis, feature-driven analysis, and tool integration. The framework is instantiated as a software comprehension environment that supports the dynamic analysis of distributed systems, multi-language systems, and systems that use binary components. The environment supports an extensive collection of dynamic views, which are derived from the runtime data collected during the execution of the software system undergoing scrutiny.

The environment is implemented as a set of integrated tools consisting of three subsystems: data gathering, data repository, and analysis/visualization. The key issues addressed by our environment include: (a) the development of a data repository for storing static and dynamic data from distributed and multi-language applications, (b) the construction of a distributed profiler that collects runtime data from components that are distributed across a network, (c) the development of a software modeling query language that facilitates data retrieval, modeling, and analysis to create software views, (d) the development of an extensive set of software views to assist engineers in the comprehension of software systems, and (e) the decoupling of the environment’s subsystems to enhance the extensibility and ease the integration of new tools.

The software views produced by our environment include feature-interaction, feature-implementation, and feature-similarity views, as well as remote-interaction, thread-interaction,
structural, and class-usage views.

To demonstrate the effectiveness of our environment, a case study involving three significant software systems was conducted (i.e., the Mozilla web browser, the Jext text editor, and the TechReport technical report archiving system).
Chapter 1: Introduction

Maintenance is an important part of a software system’s life cycle. Studies have shown that the cost of software maintenance is 50%-80% of the total system life cycle cost [59]. The relative cost of maintenance has been increasing since the 1970s. In the early 1970s, 40% of the software life cycle cost was devoted to maintenance, 55% in the early 1980s, 75% in the late 1980s, and 90% in the early 1990s [43]. The US Department of Defense spends approximately 70% of its software budget on maintenance [56]. The allocation of effort for the three types of maintenance tasks are [72]: 20% for corrective maintenance, where changes are performed to correct faults; 25% for adaptive maintenance, where changes are made in response to modifications in the environment; and 55% for perfective maintenance, where changes are made to meet new requirements. The above percentages indicate that 80% of the maintenance effort is spent on non-corrective activities. Other studies have produced similar results [59].

A reduction in the cost of corrective maintenance can be achieved by limiting the number of faults through rigorous design and thorough testing. However, limiting the number of non-corrective changes is not an option, because systems evolve through these changes in order to stay in operation and meet the evolving needs of their users. In recent years, changes in hardware and software infrastructure have become a major cause for adaptive maintenance.

One way of controlling the cost of non-corrective maintenance is to reduce the amount of effort software engineers spend on understanding the subject system. During the maintenance phase, software engineers spend more than half of their time reading source code in order to understand the system’s logic and behavior [59, 56]. Therefore, effective software comprehension tools can help reduce the high cost of maintenance.
Modern software systems are often large, distributed, written in more than one programming language, and developed from pre-built components using frameworks such as CORBA/CCM [52, 54], J2EE/EJB [75], and DCOM/COM+ [46]. The comprehension of these systems introduces new challenges. First, the source code for many of the components may be unavailable and may be written using more than one programming language [76]. Second, the components that constitute the system may not execute on a single computer but, rather, be distributed on a network of computers. Third, even if the entire source is available, many of the interactions between classes, modules, and components may not be recovered by analyzing the source code alone. This issue is apparent in large systems that rely on dynamically-linked libraries and component models; an example of such a system is Mozilla [50], which is one of the software systems that is analyzed in our case study. Fourth, although the need for human-generated code decreases when pre-built components such as EJB and COM+ are used, the overall complexity of the system increases. Specifically, many components are used partially, thus making the unused functionality part of the system [76, 78]. Also, the many layers of software infrastructure (e.g., shared facilities and libraries) of the component framework reduce the visibility of the software’s internals. The unavailability of the source code and the development of modern software systems using multiple languages have been characterized as practice patterns for architecture re-construction [69]. The use of binary components and mixed-language development are among the challenges of comprehending modern software systems.

A complementary approach to reading the source code is to perform maintenance by instrumenting the software system’s source code, exercising the pertinent features using a profiler, and then analyzing the execution traces to determine the portions of the source code that were exercised by the features. An execution trace is a sequence of runtime events (e.g., object instantiations, method invocation events) that describes the dynamic behavior of a running program. With adequate tool support, this approach enables the developer to locate the code of interest quickly. It can also reveal runtime relationships between classes and between modules, as well as thread interactions for multi-threaded systems, which are not apparent by static analysis alone.
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. After the software’s features are specified, tools can analyze the traces to produce software 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.

Among the challenges of dynamic analysis of large systems is the significant degradation in performance and the lack of profilers that are suitable for software comprehension. Furthermore, profiling becomes more challenging when analyzing distributed, multi-threaded, and multi-language software systems.

1.1 Our Research

Modern software systems are often large, distributed, written in more than one programming language, and developed using pre-built components. In our research, we have addressed these characteristics of modern software systems by developing a software comprehension framework that facilitates dynamic analysis, feature-driven analysis, and tool integration. The framework can be instantiated as a software comprehension environment that supports the dynamic analysis of distributed systems, multi-language systems, and systems that use binary components. The feature-driven analysis allows software engineers to focus their analysis on the portions of the system that are relevant to the maintenance task at hand, and to locate the portions of the source code that implement the features that are undergoing scrutiny. This selective profiling and analysis of program features greatly simplifies program understanding, especially when applied to large systems. In this dissertation, a program feature is defined as an externally visible functionality such as a use-case scenario (i.e., print or save). The architecture of the framework is loosely coupled to allow a variety of data collection and analysis tools to be integrated.
Most current software comprehension tools assume that the software is written in a single programming language and that the source code is available for analysis. These assumptions are not valid when analyzing modern software systems due to component distribution, multiple languages, and the use of binary components. Our framework addresses the distribution of components and the multi-language aspects of modern software systems by supporting the profiling and analysis of distributed systems. The framework also provides an extensible data repository to store both static and dynamic data collected from programs written in different programming languages. Modifications to the repository in order to add a new type of entity, relationship, or language only requires updating the repository’s meta-data; a change to the repository physical schema is not required. Thus, the tools that rely on the repository remain unchanged when modifications are made. In addition, we the framework has a query language to simplify the retrieval and analysis of data from the repository, as well as the creation of software views.

The architecture of the software comprehension framework consists of three subsystems: data gathering, data repository, and modeling/visualization. The framework supports static and dynamic analysis of multiple languages, distributed systems, and binary components. The framework creates three types of abstractions to represent program artifacts: entities such as functions and classes, binary relationships such as invokes and implements, and runtime events such object-create. The abstractions are common to both static and dynamic data and to all programming languages that can be modeled using these abstractions. The framework also defines a software modeling language for data retrieval and for creating software views.

Our work emphasizes three areas: (a) the instantiation of the framework as a software comprehension environment, (b) the definition and implementation of dynamic software views, and (c) an extensive case study to demonstrate the effectiveness of our environment in analyzing various types of modern software systems.
**Software comprehension environment**

The framework has been implemented as a software comprehension environment that addresses four key issues. First, the design and implementation of a single data repository for storing static and dynamic data from distributed, multi-threaded, and multi-language applications. Second, the construction of a distributed profiler that collects runtime data from components that execute across a network. The profiler has the capability of identifying *endpoints*, which are associated with method/function calls that partake in interprocess communications between distributed components. These *endpoints* are identified through the network-interceptors of the profiler, which monitor invocations to the network libraries. Third, the development of a software modeling query language (SMQL), which is the basis for data retrieval and the implementation of analyzers for creating software views. Fourth, the decoupling of the environment’s subsystems to enhance the extensibility and ease the integration of new tools. External data collection tools can be used with our environment, and data can be imported into the repository using our data import tool or any other data import tool. The implementation of our repository uses a standard SQL/JDBC database, which simplifies the importing of data and the evolution of the repository’s model. The extensibility of the analysis subsystem is supported by the implementation of SMQL filters written in Java.

The major contribution of our research is the development and implementation of a software comprehension framework that supports the creation of dynamic views to address key challenges in comprehending modern software systems.

**Dynamic software views**

The software views developed as part of our evaluation of the software comprehension environment include feature-interaction, feature-implementation, and feature-similarity views as well as remote-interaction, thread-interaction, structural, and class protocol views. The feature-interaction view highlights the dependencies between features based on the objects created and used during the execution of each feature. The interactions between features are recovered using dynamic analysis and the use of profilers that capture the runtime ob-
jects referenced during the execution of the features. The feature-interaction view is useful to assess the impact that changes made on one feature can have on other features, as well as to determine the dominant feature(s) of an application. The feature-implementation view identifies the portions of the source code that implement a feature, and it also highlights the portions of the source code that are shared between features. This view can help to identify the features that can be affected from changes made to the source code. The feature-similarity view characterizes the similarity between each pair of features. The similarity view helps engineers identify similar features and, thus, guides them to learn about the implementation of a feature by studying similar features. The similarity measure also helps engineers assess the impact of changing one feature on other features. The remote-interaction view highlights the interactions between the components of a distributed software system. The structural views, such as class-interaction and module-interaction, highlight object instantiation and interaction between objects via method invocations. Finally, the class protocol view highlights how to use an instance of a class (i.e., what order of method invocation sequences are valid).

Our key contributions with respect to the feature-based views are: (a) the feature-interaction view based on object-interaction, and (b) the method we developed to mark execution traces to identify feature, which allows the flexibility to mark any program feature at anytime during the execution of the program.

Case study

Three software systems was analyzed as part of the evaluation of the framework and its implementation. First, the analysis of the Mozilla web browser demonstrates the effectiveness of the environment to profile and analyze large software systems on the Win32 platform that are written in C/C++. Feature-implementation, feature-similarity, module-interaction, and class-interaction views are created during the analysis of Mozilla. Second, the analysis of the Jext text editor demonstrates the profiling and analysis of medium-sized Java applications. Feature-interaction, feature-implementation, and class-interaction views are created for Jext. Third, the analysis of the TechReport technical report archiving sys-
tem demonstrates the profiling and analysis of distributed systems. Feature-interaction, feature-implementation, class-interaction, remote-interaction, and thread-interaction views are created for TechReport.

1.2 Dissertation Outline

Following is an outline of the chapters in this dissertation:

- **Chapter 2 - Background**
  This chapter describes an overview of software comprehension and surveys related work. It examines research areas that are related to our research in the areas of dynamic analysis, program feature analysis, distributed systems analysis, and multiple language analysis.

- **Chapter 3 - Software Comprehension Environment**
  This chapter describes the architecture and implementation of the software comprehension framework. It describes the issues addressed in the implementation of the software comprehension environment such as the support for component distribution, multiple languages and feature analysis. Also, it describes the (SMQL) language, which is designed to query the repository and facilitate the creation of software views.

- **Chapter 4 - Dynamic Software Views**
  This chapter describes the dynamic software views that were developed during our evaluation of the framework and the its implementation. The views are dynamic and feature-driven. The base views created from the runtime data are object-interaction, call-graph and method invocation sequences views. From the base views, other views are derived such as class-interaction, module-interaction, thread-interaction and remote interaction views. The high-level views include feature-interaction, feature-implementation, and feature-similarity views.

- **Chapter 5 - Case Studies**
  This chapter describes the analysis of three software systems to demonstrate the effectiveness of our environment. Each system in the our case study demonstrates several
aspects of our research. The Mozilla web browser demonstrates the effectiveness of the environment to profile and analyze large software systems on Win32 platform written in C/C++. The Jext text editor demonstrates the profiling and analysis of medium-size Java applications. The TechReport, a technical report archiving program, demonstrates the profiling and analysis of distributed systems.

• Chapter 6 - Conclusions

This final chapter of our dissertation summarizes our research contributions and describes future research opportunities to extend our work.

• Appendix A - SMQL: Software Modeling Query Language

This appendix describes the syntax of the SMQL query language.

• Appendix B - User Manual

This appendix describes the software comprehension environment tools, including installation and usage instructions. It also describes how to write SMQL extensions in Java to create software views.
Chapter 2: Background

Software comprehension typically involves a data gathering phase using static and dynamic data collection tools, an analysis phase where the data is transformed into abstractions describing the software views, and a visualization and exploration phase.

The data gathering phase involves collecting data about the structure and behavior of the software system. There are two primary sources of program data. The first source is static data extracted mainly from the source code. Several source code analysis tools are available such as Acacia [13] for C/C++, and Chava [35] for Java. Static data about a program can also be extracted from compiled binaries and dynamically-linked libraries, as well as from pre-compiled code such Java bytecode and Microsoft.Net Intermediate Language (MSIL) [47]. The second data source is runtime data collected during the execution of the program using a profiler or a debugger. Profiling is generally accomplished either through instrumentation of the source code or through an interface provided by the runtime environment such as the JVMDI or JVMPI interfaces [70] for Java.

The analysis phase involves the transformation of the collected data into abstractions, called views, of the system structure and/or behavior. The analysis is aimed at extracting pertinent information from large amounts of data to create software views. In the literature this phase is usually referred to as the architectural recovery process. Work in this area can be classified into four categories. First, the conventional analysis methods, which focus on extracting implementation-level entities to create static and dynamic views. Examples of static views are inheritance hierarchies and module dependency graphs. Examples of dynamic views are UML sequence diagrams [38, 66] and state charts. Second, the design pattern recovery methods, which focus on identifying implementation-level design patterns (e.g., Pat System [40], Dali [34]) in the source code. Third, the subsystem identification
methods, which use clustering algorithms to group related implementation-level modules into architectural-level subsystems (e.g., Bunch [42] and ARCH [64]). Fourth, the architectural style recovery methods, which focus on extracting the overall organization of the system. The X-Ray system [45] recovers the architecture of distributed systems using static data extracted from the source code of C/C++ programs. The Dali workbench [28] supports a semi-automatic approach to identifying system architecture.

Unlike static analysis, the data collected from dynamic analysis is enormous. For example, executing a ‘Hello World’ program written in Java produces thousands of runtime events, but a partial execution of Mozilla produces millions of runtime events. The volume of data created from static analysis depends only on the system size, while the volume of data created from dynamic analysis depends on both the size of the system, the patterns of usage, and the duration of the program’s execution. The enormous amount of data makes the filtering and analysis a crucial step in the construction of useful software views.

The visualization and exploration phase involves the rendering of software views into visual forms, and the metaphors to navigate between different views. The promise of visualization is to provide program abstractions in visual forms that simplify the understanding of programs. Early work in this area was on algorithm animation [60], which is often used for educational purposes. Recent efforts have focused on visualizing the execution profiles of programs for performance, debugging, testing, and software understanding. Related work in this area can be grouped into three categories. First, content visualization, which focuses on the visual presentation of inheritance hierarchies, sequence diagrams [38], trends-based event sequences [33], as well as charts and grids [58], which are useful during debugging and performance tuning. Second, visualization techniques, which include graph drawing and layout techniques such as those supported by the dot/dotty system [26]. These techniques deal with visualization in general and are not specific to software engineering. However, much of the research is applicable to software visualization [30, 6]. Third, visual navigation techniques, which focus on the exploration of views such as fish-eye views, focus+context [30], starfield displays [2], and hierarchical views [68].
2.1 Software Views

The process of constructing software views is depicted in Figure 2.1. First, the program artifacts are extracted and stored. The artifacts represent program entities (e.g., functions, classes) and relationships between the entities (e.g., invoke, subclass), which comprises the program model. The program model is the collection of program artifacts, which includes the static data extracted from the source code and the dynamic data collected from the runtime event during the execution of the program. Then, various query and analysis tools operate on the program model to create program abstractions called software views.

Depending on the characteristics of the software system and the program model, various views can be created. There are two broad categories of software views, usually, referred to as dynamic views and static views. Static views are views that are created through static analysis, by operating on the source code artifacts, while dynamic views are views created through dynamic analysis, by operating on the runtime events collected during the execution programs.

CIA [15] is an example of an analysis system that supports static analysis for several programming languages including C/C++, Java [35] and Html [14]. Analysis and filtering capabilities in CIA are supported through the CIA query language. Examples of views that can be derived using CIA include: call graphs, inheritance hierarchies, and reachability graphs. SCED [39] is an example of an analysis system that supports dynamic analysis. This system uses runtime data to create dynamic models of object-oriented programs, which are visualized as state diagrams or state charts.
Generally, the program model can be represented as a set of graphs, where each graph illustrates certain aspects of the system. Formally, a program model \( (PM) \) is

\[
PM = \{ G_k(E_k, R_k) | E_k \subset E, R_k \subset R \}
\]

Where,

\[
E = \text{Set of all entities in the repository} \\
R = \text{Set of all relationships in the repository} \\
E_k = \text{Set of entities (nodes) in graph } G_k \\
R_k = \text{Set of relationships (edges) in graph } G_k
\]

The graphs of the views are directed and labeled (or attributed) graphs, that is, nodes and edges may have labels and other attributes associated with them. Examples of such graphs are:

- Inheritance hierarchy graphs, where nodes represent classes and edges represent the \textit{subclass} relationship. In this example, the attribute (or label) of each node is the name of the class, while the attribute of each edge is the \textit{subclass} relationship type.

- Call graphs, where nodes represent methods and the edges represent the \textit{invoke} relationship. In this view, the label of each node is the name of the method associated with the node, and the attribute of each edge is the \textit{invoke} relationship type.

Figure 2.1 shows that a software view can be constructed from the program model in the repository in two ways:

1. Directly without further analysis, that is, the resultant views are the result of queries to the repository. Queries are referred to as \textit{filters}, since their task is to retrieve a subset of the entities and relationships stored in the repository. Examples of filter views are the inheritance graphs and the call-graphs.

2. Applying a transformation on the result retrieved from a query to the repository in order to create the software view. These transformations are referred to as \textit{analyzers}.
Examples of such analyzers are feature-interaction, remote-interaction, and subsystem decomposition views.

The key difference between the two methods is that the filters operate on entities and relationships that exist in the software repository. Thus, the sets of nodes (entities) $E_k \subset E$ and edges (relationships) $R_k \subset R$ in the graph of the view are in the repository, while the analyzers transform the results of the queries into a different form that may be a graph or another visual form. If the view is a graph, the graph contains additional (derived) entities and relationships. For example, if the analyzer is a clustering algorithm, it will produce new entities, called clusters, that represent subsystems, and new relationships between the clusters.

Next, we describe the state-of-the-art of analysis techniques that are used to create graph-based software views.

### 2.1.1 Techniques for Graph-based Software Views

As described in Section 2.1, software views are often modeled as directed and attributed (or labeled) graphs. The majority of techniques for constructing software views (referred to as architectural recovery techniques) rely on graph manipulation, graph matching, pattern-matching and pattern recovery techniques.

An example of graph manipulation is abstracting the low-level views such as object-interaction graphs or call-graphs into higher-level views that depict class-interactions or module-interactions. This type of manipulation can be thought of as grouping the nodes of the graph of the view recursively into their container entities, then adding an edge between two higher-level entities if at least one relationship exists between the low-level entities. More elaborate techniques for grouping related entities (nodes) in the graph of a low-level view are clustering techniques, which cluster/group entities based on a variety of well-defined criteria. Most of the research work in the area of subsystem identification, aimed at producing high-level architectural views, relies on clustering techniques. An example of such techniques is developed by Mancoridis et al. [42]. Their techniques are implemented in a tool called Bunch. The clustering algorithm uses optimization to maximize cohesion...
between entities in the same cluster and minimize coupling between entities in different clusters. The result of the clustering process is a new higher-level view.

**Pattern-matching** techniques are used to search for certain patterns in the program model (artifacts). The process consists of two steps. First, the patterns are defined as pattern models. Second, the program model (repository) is searched for the patterns that match the defined pattern models. These techniques differ in the way they model the patterns. For example, Kramer and Prechelt [40] used the Prolog language to define the patterns; Prolog is also used as the pattern search engine. In their work, structural information is extracted from C++ header files using a commercial case tool [32].

Software views are generally represented as graphs, and **graph matching** can be used to measure the similarity between program features or components by matching the graphs that represent them. The similarity measure helps the engineer identify related features and, thus, guides him/her to learn about the implementation of a feature, by studying similar features. The similarity measure also helps the engineer to assess the impact of a change of one feature on other features in the software system. In addition, graph matching can be used to locate design patterns in the software system [63]. Sartipi et al. [63] developed a pattern-matching approach for architectural recovery using graph-matching. Patterns are defined as a graph using AQL (Architectural Query Language), then the repository is searched for matches. The resultant matches are inspected, and if needed, the pattern model is adjusted. The similarity measure in Sartipi’s approach is based on computing the edit distance between the graphs representing the pattern model and the patterns in the repository. Wong et al. [81] used a set-based similarity matching approach to measure the disparity (or similarity) between program features by comparing the basic execution blocks exercised by each feature.

**Protocol recovery** techniques are a well-studied topic in automata theory and artificial intelligence. Recently, as a result of the increased popularity of finite state machines in connection with various object-oriented design methods (*e.g.*, UML [24]) this problem has attracted the interest of the software engineering community. The aim of protocol recovery is to synthesize a finite state machine from a finite set of examples. In the area of
auto-programming, Biermann and Krishnaswamy [7] developed a system that constructs programs from example computations supplied by the user. The examples are given in a form of condition-instruction pairs, with boolean conditions on the variables. Programs can be represented as directed graphs with instructions as nodes and transitions as edges. The essential problem addressed by Biermann and Krishnaswamy is how to label instructions that appear multiple times in the sample computations, so that the end result is something more interesting than a linear directed graph.

The auto-programming system was the basis for an algorithm developed by Koskimies and Makinen to synthesize state machines from event trace diagrams [37]. A UML event trace diagram describes the order in which certain events are sent from one object to another. Motivated by the fact that event trace diagrams are not readily available for many legacy software systems, Systa and Koskimies proposed the use of runtime information as a way to generate event trace diagrams [73]. These trace diagrams can then be used with the Koskimies and Makinen algorithm to synthesize a state machine that describes the behavior of the various classes of objects of a legacy system.

Next, several more specialized research areas that are relevant to our work are described. Namely the analysis of multi-language applications, the analysis of distributed systems, and the analysis of program features.

### 2.2 Analysis of Multi-language Applications

Traditional software comprehension tools assume that the software is written in a single programming language and that the source code is available for analysis. For many modern systems these assumptions are not valid. For example, in web-based information systems, the source code typically comprises a mixture of scripting languages such as Active Server Pages (ASP), Php, Perl or Java Server Pages (JSP). In addition, the use of pre-built components, such as COM and EJB, is common. Multi-language development is popular in today’s development environments as is indicated by a survey by Evans Data Corporation [23], which shows that 60% of the enterprise developers surveyed use multiple languages.
These issues have been recognized in recent research [15, 29]. The multi-language aspect of web applications has been studied by Hassan and Holt [29]. Their approach uses source and binary extractors to perform static analysis. The extracted data is consolidated into a common fact base.

Acacia [15] supports the static analysis of several languages including C/C++, Java [35], Html [14] and ksh. In Acacia, a multi-language application could be analyzed piecemeal for each language. However, a different repository for each language is created, and, hence, queries that cross language boundaries are not possible.

There are two essential requirements for multi-language analysis: (a) the existence of data collection tools for different languages, and (b) a data model (schema) of the common repository. The second requirement has received little attention compared to the first one. The importance of data modeling has been recognized [16, 21, 55] as a key requirement for exchanging data between different tools. Koschke et al. [36] introduced a set of requirements for the intermediate representation of data for reverse engineering tools. Kullbach et al. [41] developed an integrated conceptual model for multiple languages (MVS/JCL, COBOL, PL/I and others) and a graph-based query language. Our approach differs from Kullbach’s approach in two ways. First, our data repository consists of both static and dynamic data about the programs being analyzed. Second, the data repository is extensible so that new languages can be added easily. The repository can be extended by adding (updating or removing) records to the meta-data, while the physical schema of the repository remains unchanged.

2.3 Analysis of Distributed Systems

The maintenance of distributed systems is inherently difficult because of issues such as asynchronous communication, network delays and crashes, lack of a common physical clock, and the lack of a global system state [22, 65]. In a distributed environment, system components do not share memory, and system state is distributed across multiple components running on a variety of computers and operating system platforms. Monitoring a complete thread of execution that crosses process boundaries is difficult or impossible without the help of
distributed monitoring or distributed profiling tools.

The issues pertaining to the physical distribution of components have been studied previously [9, 45]. The BEE++ system [9] uses source code instrumentation to monitor the execution of distributed systems written in C/C++, and dispatches runtime events to various distributed software comprehension tools. The X-Ray system [45] relies on the analysis of C/C++ source code to recover the architecture of distributed systems. The client-server relationships are identified using clustering techniques and clues from the source code. For many modern distributed systems, the reliance on source code may not be sufficient for an accurate and complete analysis. First, the naming and registration services, which are commonly deployed by many frameworks (e.g., CORBA, DCOM and EJB) make the identification of interrelationships between remote components impossible by just studying the source code. This is because the location of the server and the network ports are determined at runtime. Second, the source code of many components may not be available for analysis or instrumentation, which limits the portions of the system that can be analyzed.

Our approach differs from BEE++ and X-Ray in three ways. First, dynamic data is the primary data source and static data is gathered only if the source code is available. Source code is still important, because dynamic traces may exercise only portions of the system under study. In addition, the use of both static and dynamic data enhances the quality of analysis [20, 74]. Second, no instrumentation and recompilation is needed. Our distributed profiler does not use the source code, it operates on byte-code for Java and executable objects (EXE or DLL) for C/C++ and Visual Basic (VB). Third, our environment is not limited to a single programming language or framework.

2.4 Analysis of Program Features

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 [20] or a test case [80].

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 [20] 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. Eisenbarth’s approach requires starting and ending the program for every scenario. Second, the focus of our work is on the analysis of large systems, the size of the Mozilla program in our case study is over four million lines of code, while the sizes of Mosaic and Chimera programs are close to 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, which are described in Chapter 4.

Wong et al. used program execution slices to identify the portions of the code that implement a given feature or a set of features [80].

Wilde and Scully developed a technique for locating program features by analyzing the execution traces of test cases [79]. The technique uses two sets of test cases, one set that executes the feature, and a second set that does not. A comparison of the execution traces of each set is used to identify the subprograms that implement a given feature.

Chen and Rajlich [12] have developed a technique for identifying program features from an abstract system dependencies graph (ASDG). The ASDG is derived from the source code, and it describes source code entities (e.g., procedures, types, variables) and the relationships between them. The identification of features is performed manually by examining the graph.

### 2.5 Summary

This chapter presents an overview of software comprehension and surveys related work. It introduces the software comprehension process by which software views are created, and it discusses the state-of-art of the architectural recovery techniques used to recover high level views from program artifacts. It examines specific research areas related to our research, mainly in the areas of dynamic analysis, program feature analysis, distributed systems
analysis, and multiple language analysis.

Based on the current practice of software comprehension discussed in this chapter, we focused our research on dynamic analysis in order to address the following challenges:

- **Feature-driven analysis.** Realizing the importance of identifying program features, and mapping those features to source code, we have made program features the center of our research. We incorporated a trace-marker in our profilers to allow an engineer to associate a feature with an execution trace during the execution of a feature. In addition, the trace-marker can be used to mark execution traces of a distributed system. In our research, we constructed various feature-based views such as feature-interaction, feature-implementation and feature-similarity views.

- **Remote-interaction.** Many of today’s systems are distributed, and the analysis of such systems requires distributed profilers to collect runtime data and perform analysis that will recover the remote-interactions between the components of a distributed system. We have implemented network interceptors with our profilers (JvProf for Java and Wdbg for Win32) to identify invocations that cross process boundaries such as calls to network libraries (e.g., winsock). In addition, we have implemented a logical time server that the profilers can use to adjust their logical time, so that the events from components distributed across a network can be ordered correctly.

- **Extensible data repository.** To allow the storing of static and dynamic data from programs written in different programming languages, we have developed a data repository that supports a meta-data model to represent multiple programming languages. The modifications to the repository in order to add a new entity, relationship, or a new language would only require updating the meta-data; a change to the database schema would not be required. Thus, the tools that rely on the repository remain unchanged when modifications are made on the repository’s meta-data. In addition, we have developed a query language, called SMQL, to simplify data retrieval from the repository as well as the creation of software views.
Chapter 3: Software Comprehension Environment

We present an approach to address the challenges of analyzing and understanding modern software systems that are distributed, written in more than one programming language, and developed/built using binary components. The approach involves the development of an environment that supports dynamic and static software analysis, and the use of the environment to construct software views that capture the structure and behavior of the system.

Figure 3.1 illustrates the architecture of the software comprehension environment. The main subsystems are: data gathering, repository, and analysis/visualization subsystems.

The **data gathering** subsystem defines the interfaces and data formats of the data collection tools (*i.e.*, static analyzers and dynamic profilers). At the core of this subsystem is the distributed profiler, which consists of three basic components. The first component is a local profiler that is attached to each running process. The second component implements event ordering service (*i.e.*, logical time server), which is essential since the components of a distributed system do not share a common clock. The third component is a data collection manager that receives data from the local profilers.

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 universal schema is a composite model of both data and meta-data models along with a query and data manipulation language. The repository is manipulated using standard SQL and is accessed using either SQL or our own SMQL (described in Section 3.3). SMQL integrates the data repository with the analysis and visualization subsystem. External data collection tools can be used with our environment, and data can be imported into the repository using
Figure 3.1: Architecture of the software comprehension environment

our data import tool or any other data import tool. The implementation of the repository uses a standard SQL/JDBC database, which facilitates the importing of data as well as the extension of the repository and its meta-model. The extensibility of the analysis subsystem is done through the implementation of new SMQL filters, which are written in Java.

The analysis and visualization subsystem is responsible for the creation and visualization of software views. The subsystem has three components: conceptual modeling, visualization, and navigation. The conceptual modeling component defines the set of filters and analyzers for deriving software models. The models are functions of data and context. The data part of the model is the result of a query, and the context is the meaning associated with the data. The visualization component is responsible for the presentation of the models, and the navigation component is responsible for exploration of the views.

The remainder of the chapter describes the data gathering, data repository, and analysis and visualization subsystems in detail.

3.1 Data Gathering

The architecture of the data gathering subsystem is influenced by four design goals. The first goal is to support multiple languages. The second goal is to gather static and dynamic data, because both kinds of data are needed for a comprehensive analysis of the structure and
behavior of software systems. The third goal is to decouple the data collection component from the rest of the environment’s subsystems. The tight-coupling between data collection tools and the rest of the tools is a common drawback of many popular tools [27]. The final goal is to support commonly used scripting languages and component-based frameworks.

Figure 3.2 illustrates the architecture of the data gathering subsystem. The language-dependent data extractors are responsible for extracting program artifacts from the software’s source code or execution. The data collection adapters encode source code facts and runtime events and route them to the data collection manager. The adapters provide a common interface for transporting the collected data as serialized Java objects or as XML messages. The time server provides a shared logical clock to enable the correct ordering of the runtime events when profiling distributed systems. The data collection manager routes data to the data repository and provides access to the language meta-data models.

3.1.1 Remote Interaction

A key requirement for analyzing distributed systems is the identification of remote interactions between components. The profiler must be able to distinguish between local interactions and remote ones. Remote interactions occur when a component invokes special functions that result in a request/reply messaging protocol between two distributed compo-
nents. For example, in Java, the special functions include the \texttt{read()} and \texttt{write()} methods of the \texttt{SocketInputStream} and \texttt{SocketOutputStream} classes respectively. In C/C++, the special functions include the \texttt{recv()} and \texttt{send()} functions of the \texttt{winsock.dll} library.

To support the analysis of distributed systems, the data collection and repository sub-
n-systems model the concept of communication \textbf{endpoint} entities, and the \textbf{connects} relationship between endpoints.

Figure 3.3 shows portions of a UML event sequence diagram of distributed components of a distributed system. In the diagram the solid diamonds highlight \emph{endpoints}, the gray thick arrows show the \emph{connects} relationships between the components. A method from the \texttt{userClass} class invokes the \texttt{write()} method of the \texttt{SocketOutputStream} class to send a message to the \texttt{userModule} module. The \texttt{userModule} uses the \texttt{recv()} function (in the \texttt{ws2_32.dll} library) to read the message. The \texttt{userModule} uses the \texttt{send()} function to send a reply to \texttt{userClass}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.3.png}
\caption{Communication Endpoints}
\end{figure}

An endpoint is defined as a tuple of four elements: network address of the local host, local network port, network address of the remote host, and the remote network port. The \emph{connects} relation between two endpoints \textit{A} and \textit{B} is valid if all of the following conditions are satisfied (see Figure 3.4):

\[ EP_{\text{localHost}}(A) = EP_{\text{remoteHost}}(B) \]
\[
EP_{\text{localPort}}(A) = EP_{\text{remotePort}}(B) \\
EP_{\text{remoteHost}}(A) = EP_{\text{localHost}}(B) \\
EP_{\text{remotePort}}(A) = EP_{\text{localPort}}(B) \\
EP_{\text{Time}}(B) - EP_{\text{Time}}(A) < \text{Threshold} \\
EP_{\text{Time}}(B) > EP_{\text{Time}}(A)
\]

Where,

\(EP(A)\) : An endpoint from application \(A\)  \\
\(EP(B)\) : An endpoint from application \(B\)  \\
\(EP_{\text{localHost}}(A)\) : The domain name or IP address of the machine hosting application \(A\)  \\
\(EP_{\text{localPort}}(A)\) : The TCP/IP port number for application \(A\)  \\
\(EP_{\text{remoteHost}}(B)\) : The domain name or IP address of the machine hosting application \(B\)  \\
\(EP_{\text{remotePort}}(B)\) : The TCP/IP port number for application \(B\)  \\
\(EP_{\text{Time}}\) : The logical time-stamp when the endpoint was detected  \\
\(\text{Threshold}\) : The acceptable time difference between the detection of the  

two endpoints, ideally its value is one logical-time tick

The above conditions ensure that the correct pair of endpoints are associated to establish the remote interaction between two distributed components. The endpoint entity is identified by the distributed profiler using network interceptors, while the connects relation is determined by analyzing the event trace.

**Figure 3.4: Endpoint entity and connect relation**

Modeling the runtime interactions between more than one application requires profilers
that can intercept network calls (e.g., `socket` or `SocketStream` calls) to determine the communication endpoints between the participating distributed components.

We implemented *network-call interceptors* in **Wdbg** [61] (for Win32 applications) and **jvprof** [66] (for Java applications) local profilers. Running either or both profilers in conjunction with the data collection manager forms a distributed profiler for collecting runtime data from distributed applications written in Java and Win32 C/C++/VB.

The Java local profiler (**jvprof**) uses the JVMDI and JVMPi interfaces [70]. The profiler watches for method call events from objects that are instances of the `SocketInputStream` and `SocketOutputStream` classes. The endpoint parameters are determined by reading the field values of the `SocketImpl` class. The instance of `SocketImpl`, which is declared in the `SocketInputStream` and `SocketOutputStream` classes, holds the values of the remote and local network addresses and the port numbers. Note that Java RMI and EJB use these classes to implement remote invocations.

Figure 3.5 illustrates a typical configuration of the profilers to construct a distributed profiling environment. In addition to identifying remote-interactions, a user can mark traces of execution using the mark-trace wizard. The traces can be marked as distributed to instruct the time-server to pass the trace-name from one program to another. This capability allows users to profile features that cross the boundary of a process and identifies the code that is executed at the client and the server side for the feature.

### 3.1.2 Feature-driven Profiling

In a typical 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 a word processing application. The developer studies the source code to locate the portions that are related to the ‘print’ feature. For many software systems, this task is difficult and time consuming, since the implementation of a feature may involve multiple classes or modules.

A developer can take an alternative approach to this maintenance task by instrumenting the application’s code, and exercising the subject feature using a profiler, and then ana-
Figure 3.5: Distributed profiling

Analyzing the execution traces to determine which portions of the code were exercised by the feature. An *execution trace* is a sequence of runtime events (e.g., object creation/deletion, method invocation, thread creation/termination, · · ·) that describes the dynamic behavior of a running program. With adequate tool support, this approach is more effective, because it reduces the complexity of the task by allowing the developer to locate the code of interest quickly. This approach is suitable for many practical settings, since change requests are usually written in plain language with explicit references to identifiable program 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.

The software comprehension environment is constructed to facilitate selective profiling and, hence, the execution of selected features during program execution of a program. The approach is centered around the concept of marked execution traces, which developers use to define program features. In this dissertation, a *feature* is defined as a functionality, or a usage scenario of a program, that can be represented as a sequence of runtime events. Features are
specified by the developer 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. The trace-marker utility is implemented in the dynamic profilers, and the trace-name is made an attribute of the runtime events and relationships to facilitate feature-driven analysis of program runtime data.

3.2 Data Repository

The schema of the data repository comprises three models. The first is the Language Definition model, which defines the entities and relations for each programming language. The second is the Program Data model, which stores the data gathered (entities, relations, and runtime events) about the programs under study. A runtime event is defined as a relation with a runtime context. The runtime context includes the time when the event occurred, the process, and thread that generated the event. The third model is defined as a set of database views to simplify querying the repository. The SMQL query language (described later in this section) uses these database views.

3.2.1 Language Definition Model

The language definition model is a graph, where nodes define programming language entities, and edges define relations between entities. Formally,

\[
\text{GeneralModel} = \text{Graph}(E_g, R_g)
\]

Where,

\[E_g\] is the set of all entity types supported by the repository.

\[R_g\] is the set of all relation types supported by the repository.

Each element in \(R_g\) is a tuple of three basic elements: \(\text{RelationType}, \text{SourceEntity}, \text{and TargetEntity}\). A specific programming language model (e.g., Java) is defined as the projection of the general model over the set of entities and relations supported by the language.
For example, the Java language can be defined as:

$$Model(Java) = Graph(E_{java}, R_{java})$$

Where,

- $E_{java} \subseteq E_g$ is the set of all entity types supported by the Java language
- $R_{java} \subseteq R_g$ is the set of all relation types supported by the Java language, such that source and target entities $\in E_{java}$

In addition to programming entities and relations, the model defines the set of applicable runtime events and their attributes for each defined language. The graph in Figure 3.6 illustrates the Java language data model. Note that the edges to and from the endpoint entity (diamond shape) are represented in dotted lines to emphasize that these edges are determined by analyzing the event trace and are not directly extracted from the source code.

A new language model can be instantiated from the general model or from an existing language model. For example, a C++ model can be defined from the Java model as:

$$Model(C++) = Graph(E_{c++}, R_{c++})$$
where,
\[
E_{c++} = E_{java} \cup \{\text{template, struct, typedef, function}\} - \{\text{interface, package}\}
\]
\[
R_{c++} \subset R_g \text{ such that source and target entities } \in E_{c++}
\]

The relational schema of the language model is depicted in Figure 3.7. The language definition can be extended by adding a new language or modifying an existing language without the need to change the physical schema of the repository. Extensions are performed by updating the content of the dictionaries (tables prefixed with D) of the repository. The key benefit is that the data collection and analysis tools that rely on the repository will not need to be modified whenever a new language entity or relationship is defined. The language model is linked to the program data model to maintain the referential integrity of the repository.

Figure 3.7: Language definition model
3.2.2 Program Data Model

The program data model defines the schema for storing the data collected by the static and dynamic analysis tools. The valid entities, relations, and events for each supported language are determined by the language definition model described in the previous section.

Each entity is stored as a tuple:

\[
\text{Entity} = (\text{EntityType}, \text{Attribs}, \text{UsrAttribs})
\]

where,

\[
\text{EntityType} \in \{\text{File}, \text{Package}, \text{Module}, \text{Class}, \text{Interface}
\]

\[
\text{Component}, \text{Method}, \text{Function}, \text{Variable}, \text{Field}, \cdots\}
\]

\[
\text{Attribs} \subset \{\text{Name}, \text{Signature}, \text{Container}, \text{Project}, \text{SourceFile}, \cdots\}
\]

\[
\text{UsrAttribs} = \text{User defined attributes}
\]

The supported values of \text{EntityType} are defined in the D\_EntityTypes table of the language model. The values of \text{Attribs} are the common attributes for all entities. The
UsrAttribs are name/value pair of entity attributes that are not defined in the language model. If a data collection tool collects data that cannot be stored directly in the Entity table (e.g., a new entity attribute that is not defined), then the data is stored in the EntityAttrib_Usr table as a name/value pair, which can be queried as any other attribute. Each relation is stored as tuple:

\[
Relation = \{ \text{RelationType}, \text{SourceEntity}, \text{TargetEntity}, \text{TraceName}, \text{UsrAttribs} \}
\]

\[
\text{RelationType} \in \{ \text{Inherits}, \text{Contains}, \text{Declares}, \text{Implements}, \text{Creates}, \text{Invokes}, \text{Accesses}, \text{Modifies}, \ldots \}
\]

\[
\text{UsrAttribs} = \text{User defined attributes}
\]

The supported values of RelationType are defined in the D_RelationTypes table of the language model. The SourceEntity and TargetEntity are the source and target entities participating in the relationship. The UsrAttribs are name/value pairs of additional relationship attributes. The TraceName is the value selected by the engineer to label a feature or a use-case during the execution of the program, this value is obtained from the trace-marker utility. Each runtime event is stored as tuple:

\[
Event = \{ \text{EventType}, \text{SourceEntity}, \text{TargetEntity}, \text{Attribs}, \text{UsrAttribs} \}
\]

\[
\text{EventType} \in \{ \text{Process-start}, \text{Process-end}, \text{Thread-start}, \text{Thread-end}, \text{Method-entry}, \text{Method-exit}, \text{Object-create}, \text{Object-free}, \ldots \}
\]

\[
\text{Trace-start}, \text{Trace-End}, \ldots \}
\]

\[
\text{Attribs} \subset \{ \text{Project}, \text{Project-id}, \text{Thread-id}, \text{Timestamp}, \text{LogicalTime}, \text{object-reference}, \text{TraceName}, \ldots \}
\]

\[
\text{UsrAttribs} = \text{User defined attributes}
\]

The supported values of EventType are defined in the D_EventTypes table of the language model. The SourceEntity and TargetEntity are the source and target entities describing the event. The SourceEntity value is always required, while the TargetEntity value is
optional and depends on the type of event. The endpoint event is a specialization of the runtime event.

3.3 SMQL: Query Language

Although the repository can be queried using SQL, designing queries for comprehending software systems using SQL is cumbersome. In addition many of the queries that are of interest to a software engineer, for example queries that involve the transitive closure of a relation, are not supported directly by SQL (e.g., a query to determine reachability between functions). To overcome this limitation, we developed a higher-level query language called SMQL (Software Modeling Query Language). 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, which computes forward and reverse reachability, as well as binary operators such as union, intersection and join. SMQL is similar to grok [31] for manipulating binary relational algebra expressions. Unlike grok, SMQL can be extended to support additional operations, which are implemented using Java.

To introduce SMQL, we use an inheritance hierarchy view example (Figure 3.9). The SMQL code computes the inheritance hierarchy rooted at either the org.mortbay.http.HttpListener or org.mortbay.http.HttpHandler classes or interfaces. First the entities of interest are specified in the EntitySet block, then the closure function is invoked to compute the reverse reachability of any class to the HttpListener and HttpHandler interfaces. The result of the execution is two sets: \( C \) is an entity set with two elements \{HttpListener, HttpHandler\}, and a relation set InheritanceForest, which holds the result returned by the closure function. The last line in the code invokes the output.dot function to store the relation set as a dot-graph shown in Figure 3.10. In the graph, the package prefix org.mortbay is removed from the node names in the graph. In the graph, the solid lines represent subclass relations, and the dotted lines represent implement relations.

In the SMQL code listing in Figure 3.9, the EntitySet declares an entity set, the type
// Example 1: Inheritance view
// Declare an entity set "C"

EntitySet C
{
/**
Get Inheritance forest rooted at org.mortbay.http.HttpListener
or org.mortbay.http.HttpHandler
**/
type = {"interface", "class"};
include (name) = { "org.mortbay.http.HttpListener",
                 "org.mortbay.http.HttpHandler" };
include (project) = { "jetty-static" };
}

// Use closure to get all subclasses to determine the inheritance
// forest
InheritanceForest = closure(null, C, { "implement", "subclass"} ) ;

// Save the result as a dot-graph
output_dot ("c:\temp\Inheritance.dot", InheritanceForest );

Figure 3.9: SMQL: Inheritance example

defines the type of entities to be included in the query, and the include clause defines a
filter on the entities based on the entity name and the project in which the entity belongs.
The valid attributes of the include and exclude filters are the attributes of the entities, events, and relations of the repository.

The closure function returns a set of relations. The semantics of the closure function
is as follows:

closure(Source, Target, RelationTypes)

where, Source is the source entity-set, Target is the target entity-set, and RelationTypes is
the set of relation types over which the closure is computed. If the Source is an empty
set (null), the closure function computes the reverse reachability from all entities in the
Target entity set. In the example of Figure 3.9, the reverse reachability from any class to any
member of the entity set C over the {subclass, implement} relation types produces a relation
set that represents the inheritance forest root at either HttpListener or HttpHandler. If
the Target entity set is empty, then closure computes the forward reachability from all
entities in the Source entity set. If both are provided, closure computes the reachability from any entity in the Source entity set to any entity in the Target set. If both Source and Target are empty sets, then the function will fail and returns an error.

A summary of the capabilities of the SMQL language is outlined below:

- SMQL is used to define and query the repository for program entities, relationships and runtime events using the EntitySet, RelationSet and EventSet blocks respectively. Zero or more include or exclude filters can be defined in each block. All results are presented to the user in the SMQL result window of the environment. Each resultant item has a default visualizer attached to it. An entity set is visualized as a decomposition hierarchy, and a relation set is visualized as a graph, which may contain multiple views, and an event set is visualized as a list. In Section 3.4, we describe in detail the visualization of these sets.

- SMQL provides basic set operations such as union (+), intersections (*), and difference (-).
• SMQL supports the integration of new analyzers, written in Java. These analyzers are imported into SMQL as functions.

• SMQL provides basic built-in functions such as `closure`, `composition`, `output_dot` and `output_xml`.

The benefits of SMQL are:

1. Provides a common mechanism for query processing and data retrieval.

2. Provides a platform for integrating external analysis and visualization tools (e.g. Bunch clustering tool, dot graph layout tool).

3. Provides a common platform for viewing query results.

4. Provides a common approach for developing software views, namely user-defined filters, which are implemented as extensions written in Java or as wrappers of other external tools [62].

5. Provides a framework for developing software views.

6. Simplifies the importing and exporting of data from the repository.

The complete SMQL syntax is described in Appendix A.

### 3.3.1 SMQL Objects

SMQL deals with five types of variables, called SMQL objects. These objects are outlined below:

• Basic objects. These are objects created from the three types of declarations:

  – `EntitySet` constructs an entity set based on a set of entity types and zero or more `include` or `exclude` filters. SMQL builds an equivalent SQL statement and submits the query to the repository to retrieve the entities specified in the `EntitySet` construct.
- **RelationSet** constructs a relation set based on a set of one or more *include* or *exclude* filters. SMQL parser builds an equivalent SQL statement and submits the query to the repository to retrieve the relations specified in the *RelationSet* construct.

- **EventSet** constructs an event set based on a set of zero or more *include* or *exclude* filters. SMQL parser builds an equivalent SQL statement and submits the query to the repository to retrieve the events specified in the *EventSet* construct.

- Composite objects. This type is a set, where each element is either an *EntitySet*, *RelationSet*, *EventSet* or a composite object. This type is usually constructed and returned from the SMQL analyzers to group the results of the analyzers into a single SMQL object.

- Everything else, which can be a set of strings, a simple string, or a reference to a Java object.

After the SMQL code is executed, the SMQL parser presents the results of the query as a tree. The user can then use the tree to view each of the SMQL objects. For example, the sample code in Figure 3.11 will produce the tree shown in Figure 3.12.

### 3.3.2 SMQL Query

**EntitySet**

The declaration of the *EntitySet* is described below:

```plaintext
EntityDecl ::= EntitySet Identifier
{
    [caption = String;]
    type = StringSet;
    (IncludeFilter) *
}
```
// SMQL Code

// Simple String set
myStr = {"String 1", "String 2");

// Declare an entity sets

EntitySet Entity1
{
    type = {method} ;
    include(project) = {"Java-Project"};
}

EntitySet Entity2
{
    type = {method, function} ;
    include(project) = {"C++-Project"};
}

// Declare an Relation sets

RelationSet Relation1
{
    include(invoke) = { Entity1 -> Entity1 };
}

RelationSet Relation2
{
    include(invoke) = { Entity2 -> Entity2 };
}

// Declare an Relation sets

EventSet Event1
{
    type = {"method-entry", "method-exit"};
}

// Apply “MySMLQFilter” filter
FilterResults = MySMLQFilter(Event1);

Figure 3.11: SMQL code example
SMQL Objects

Entity Sets
- Entity1
- Entity2

Relation Sets
- Relation1
- Relation2

Events Sets
- Event1

FilterResults
- Relation Sets
  - Relation3
- Entity Sets
  - Entity3

Others
- MyStr
  - String 1
  - String 2

Viewed as hierarchy/tree based on the entity containment relationships

Viewed as graphs, and can be clustered.

Viewed as a list

Figure 3.12: SMQL result tree
In the EntitySet declaration block, the **caption** is an optional attribute that specifies a brief description of the entity-set. This caption is used for readability purposes. The **type** attribute specifies the set of entity-set types to be included in the query. The valid types are the entity types defined in the D_EntityType table of the repository. The IncludeFilter declaration specifies zero or more include or exclude filters. Each filter specifies an entity attribute and the desired values. The declaration is defined as:

\[
\text{IncludeFilter} ::= (\text{Include}|\text{Exclude})(\text{attribName}) = \text{Expr};
\]

The expression \(\text{Expr}\) of the IncludeFilter is always evaluated to a set of strings, which specifies the desired values for the specified attribute.

**RelationSet**

The declaration of the RelationSet is described below:

\[
\text{RelationDecl} ::= \text{RelationSet} \text{ Identifier} \\
\{ \\
\text{[caption = String]} \\
\text{(IncludeFilter)} * \\
\text{(EntityDecl)} * \\
\text{(RelationDefs)} + \\
\}\]

The IncludeFilter declaration specifies zero or more include or exclude filters. These filters are similar to those for the EntitySet declaration. The actual relation is specified in the RelationDefs declaration.

\[
\text{RelationDefs} ::= (\text{Include}|\text{Exclude})(\text{relationType}) = \\
\{ \text{Identifier} \rightarrow \text{Identifier} (, \text{Identifier} \rightarrow \text{Identifier}) * \} ;
\]

In the above declaration each identifier specifies an entity set that is already defined, either inside or outside of the RelationSet block. The identifier to the left of the ‘\(\rightarrow\)’ defines the
source entity-set and the identifier to the right of ‘->’ defines the target entity-set. The EntityDecl declaration allows the definition of an entity set inside the RelationSet block. Each entity set inside a RelationSet block is declared the same way as described above. The difference between an entity set declared outside rather than inside the RelationSet block is that the results of the entity set declared inside the RelationSet block are not retrieved independently as entity sets, and thus, are not included in the SMQL result tree.

**EventSet**

The declaration of the EventSet is described below:

\[
\text{EventDecl} ::= \text{EventSet} \quad \text{Identifier} \\
\quad \quad \{ \\
\quad \quad \text{[caption = String;]} \\
\quad \quad \text{type = StringSet; } \\
\quad \quad \text{(IncludeFilter) * } \\
\quad \quad \} \\
\]

The declaration of an event set is similar to that of an entity set. It includes the specification of the event types and zero or more include or exclude filters.

### 3.3.3 SMQL Filters

In SMQL, software views are created as filters. An SMQL filter is a Java implementation of the IFilter interface. These filters are invoked as functions from the SMQL code. There are two types of SMQL filters: local and global. Local filters must be declared before they are used in the SMQL code, using the import construct. Global filters are loaded when the SMQL parser is started and are available to all SMQL code. Any used-defined filter can be made global by adding it to the SMQL configuration file. Currently, the following filters are implemented:
• **Closure.** The closure function returns a set of relations. The syntax of the closure function is as follows:

\[
\text{closure}(\text{Source}, \text{Target}, \text{RelationTypes})
\]

where, \( \text{Source} \) is the source entity-set, \( \text{Target} \) is the target entity-set, and \( \text{RelationTypes} \) is the set of relation types over which the closure is computed. If the \( \text{Source} \) is an empty set (null), the closure function computes the reverse reachability from all entities in the \( \text{Target} \) entity set. If the \( \text{Target} \) entity set is empty, then closure computes the forward reachability from all entities in the \( \text{Source} \) entity set. If both are provided, closure computes the reachability from any entity in the \( \text{Source} \) entity set to any entity in the \( \text{Target} \) set.

• **Composition.** The syntax of the composition filter is as follows:

\[
\text{composition}(E_1, E_2, \cdots)
\]

Where, \( E_k \) is an entity set. The composition computes the union of all the entity sets, and then computes the composition hierarchy based on the containment relationship and returns the result as an entity set. This filter is always invoked when an entity-set is visualized to present the entity-set in a form that is familiar and easy to comprehend. For example, methods are grouped in their container classes, and classes are grouped in their container package (for Java) or module (for C/C++), and modules or packages are grouped in their container application.

• **Callgraph.** The Callgraph takes an event set containing method-entry and method-exit events and produces a relation set representing the call-graph of the execution trace represented by the event-sets. This function also searches for remote-interactions if the event set includes endpoint events. The syntax of Callgraph is as follows:

\[
\text{callgraph}(E_1, E_2, \cdots)
\]

Where, \( E_k \) is a set of events. The callgraph first computes the union of the event sets and then converts the results of the union into a relation set.
• **AnalyzeEvents.** The **AnalyzeEvents** function takes one or more event sets, and computes the union of all sets, then computes several views from the events. The syntax of **AnalyzeEvents** is as follows:

\[
\text{AnalyzeEvents}(E_1, E_2, \cdots)
\]

Where, \( E_k \) is a set of events. The **AnalyzeEvents** filter converts the results of the union into a composite set. Each element of the composite set represents a set of dynamic views. The views computed by this SMQL filter are:

- **Object-interaction** views are computed per thread. Each view is a relation set that is visualized as a graph.

- **Invocation sequences** views are computed per thread. Each view is a relation set that is visualized as a graph.

- **Mark traces** views are computed per marked-trace (or feature). Each view is a relation set that is visualized as a graph. In addition, this view include two additional views: feature-interaction and feature-implementation views, which also are visualized as graphs.

- **Thread-interaction** views are computed using the objects shared between concurrent threads.

- **Distributed interaction** views are computed by associating object interactions (or call-graphs) of two programs using a pair of connected endpoints.

• **FeatureSimilarity.** The **FeatureSimilarity** function takes one or more event sets, and computes the union of all sets, constructs the call-graph(s) for each feature, then computes the similarity matrix between the features. The syntax of **FeatureSimilarity** function is as follows:

\[
\text{FeatureSimilarity}(E_1, E_2, \cdots)
\]

Where, \( E_k \) is a set of events.
• **MethodSequences** computes the method sequences from a set of one or more event set. The output of this function is used to produce the class usage protocol.

• **output_dot** function stores a relation set as a graph in dot.

• **output_xml** function stores relations, entity, or event sets as an XML file.

In Section B.5, we describe, in detail, how to implement the IFilter interface to create analyzers using an implementation example of IFilter interface.

### 3.4 Analysis and Visualization

The **Analysis and visualization** subsystem (Figure 3.13) is responsible for the creation and visualization of software views. The subsystem has two components: conceptual modeling and visualization. The conceptual modeling component defines the set of filters and analyzers for deriving software models. The models are a function of data and context. The data part of the model is the result of a query, and the context is the meaning associated with the data. The visualization component is responsible for the presentation of the models.

![Figure 3.13: Analysis and visualization subsystem](image-url)
The modeling involves a selective data retrieval of the data of interest from the data repository, and then further analysis using SMQL filters. The selected data is defined, using SMQL, as sets of entities, relationships, and events. Further analysis on these sets may involve simple operations such as union, intersection, difference, or more advanced operations. The outcome of the data retrieval and analysis process (the result of executing SMQL code) is one or more of the following objects:

- Set of entities (EntitySet).
- Set of relationships (RelationSet)
- Composite set, where each element is either an EntitySet, RelationSet or a composite set.
- Everything else, which can be a simple string or a reference to a Java object.

These results are presented to the user as tree view, which the user interacts with to visualize each of above objects.

The process of visualizing objects is illustrated in Figure 3.14. When a user selects an object for visualization, the object is first checked to see if it has a visualizer attached to it. If a custom visualizer is attached to an object, then an instance of the visual style is created and the selected object is passed to the visualizer, otherwise the default visualizer is invoked. The default visualizer visualizes only entity sets and relationship sets. Entity sets are visualized as a containment tree (e.g., a class-node has child nodes of methods and fields). Relationship sets are visualized as graphs. The visual-style of nodes and edges is customizable so that different types of nodes can have distinct shapes. The visualizer produces three or more views for each relationship set:

- Entity containment tree of all entities participating in the relationship.
- A detailed graph, where each node represents an entity.
- A summarized graph, where each node is the container of entities with the same container entity.
• Clustered graphs of either the detailed or the summarized graph. The clustering process is initiated by the user.

The default visualizer uses the *dot* system [26] for graph layout, and uses the clustering features of the *bunch* system [42] for graph clustering. The navigation between views is accomplished by setting the ‘drill-in’ property of the nodes (entities) and edges (relationships) of the graph to the entity set or relation set.

![Figure 3.14: Visualization process flow](image)

User-defined analyzers implement the *IFilter* interface. We refer to instances of the class that implements the *IFilter* interface as analyzers. User defined visualizers implement the *ISoftView* interface. The association of a visualizer to an analyzer is performed using SMQL in one of two ways. First, the ‘sf.visualizer’ property of an SMQL object (e.g., RelationSet) is set to a Java-class name that implements the *ISoftView* interface in the SMQL code as show in Figure 3.15. Second, the ‘sf.visualizer’ property is set inside by the analyzer to the class name or to the object reference of an instance of the visualizer. This option allows the user to create the visualizer and initialize it as needed.

For example, executing the SMQL code shown in Figure 3.15 will produce the results shown in Figure 3.16. In the example, the *KwicEvents* set is converted into a call-graph using the *Callgraph* SMQL filter, the *KwicEvents* set is cloned to produce an identical copy stored in *clonedList* set, then a sequence diagram visualizer is attached to the *clonedList*
// Example 2: Attach a visualizer to a relationship set

// Define the set of runtime events

EventSet KwicEvents
{
    type = {
        "method-entry",
        "method-exit",
        "thread-start",
        "thread-end" );
    include {project} = {"kwic-rt"};
}

// Use the Callgraph() filter to convert the events into a
// callgraph (as Relation Set)
GC = Callgraph(KwicEvents);

// Clone the eventSet (making a new copy)
clonedList = clone(KwicEvents);

// Attach the SequenceDiagram Visualizer to the clone event set
clonedList["sf.visualizer"] = "serg.sc.sd.ui.SceEventSequVisualizer";

---

Figure 3.15: SMQL: Attaching a user-defined visualizer to an event set

If the user chooses to view the call-graph, the tool will show the results as a graph using the default visualizer as shown in Figure 3.17. The default visualizer for an event set is a list, therefore, if the user selects the KwicEvents set, the result will be shown as a list as shown in Figure 3.18. However, if the user selects the clonedList event set, the tool will use the attached visualizer to show the result, which in this case a sequence diagram as shown in Figure 3.19.

### 3.5 Summary

In this chapter we described the architecture of the software comprehension environment. The environment addresses the unique challenges for comprehending modern software systems such as the dynamic analysis of distributed systems, the analysis of multi-language applications, and the selective analysis of program features. Our solution includes a distributed profiler with network interceptors that identifies communication endpoints, a com-
Figure 3.16: SceTool: SMQL result tree
Figure 3.17: SceTool: Default visualizer for Relation-Set
Figure 3.18: ScéTool: Default visualizer for Event-Set
Figure 3.19: SceTool: Sequence diagram visualizer for Event-Set
mon repository for storing static and dynamic data from multiple languages, and the SMQL language for simplifying queries that cross multiple application and programming language boundaries.

Our contribution to the state-of-the-art is the design and implementation of a software comprehension environment, which addresses several key issues pertaining to the analysis of modern software systems. These issues are: the analysis of program features, the identification of remote-interactions in distributed systems, and the support of the static and dynamic analysis of software systems that are implemented using multiple programming languages.
Chapter 4: Dynamic Software Views

This chapter describes the software views that capture the dynamic behavior and structure of a software system. These views are derived from the execution traces of a program. There are two broad categories of views: static and dynamic views. The dynamic views are abstractions that describe the runtime behavior of a software system. The static views are abstractions that describe the implementation of the system. The main difference of dynamic and static views is that dynamic views are derived from runtime data collected during the execution of the program and not from the source code, while static views are derived primarily from the source code.

Figure 4.1: Hierarchy of views

Figure 4.1 illustrates a taxonomy of the software views and also highlights the required features of the profiler to create certain types of views. For example, to derive remote-interaction views, a distributed profiler is needed. In addition, to derive feature-level views
the profiler must be capable of labelling each execution trace with the name of a feature. In the taxonomy, there are different levels of abstraction. The lowest level represents the runtime events that are generated during the execution of the program. The second level of abstraction includes runtime object interactions and the method-level relationships (e.g., call-graphs and method sequence invocations). This level is the basis for higher-level views. The third level includes higher-level entities such as classes and modules, and the relationships between them. The highest level of the hierarchy represents program feature-level views. In this context, a program feature refers to an externally visible functionality of a program, and is identified as marked-traces. The set views are outlined below:

1. **Call-graph view.** This view is constructed from the execution traces of the program. Method invocation events (method-entry and exit) are used to derive the call relationships between the methods of a program.

2. **Object-interaction view.** This view is constructed from the execution traces of the program. The derived relationships are based on object creation and object usage events. It serves as the basis for higher level views such as the class-interaction and feature-interaction views.

3. **Method invocation sequence.** This view is constructed from the execution traces of the program. Method invocation events (method-entry and exit) are used to derive the relationships. This low-level view identifies the order in which methods are invoked against instances of class. This view serves as the basis for the class protocol view, in which all method invocation sequences from instances of a class are processed together to devise the protocol view of the class.

4. **Class-interaction view.** This view is an abstraction of the object-interaction view, where sets of objects are represented by their corresponding classes. However, the interactions between the classes in this view represent the dynamic relationships derived from the object-interaction view.

5. **Module-interaction.** This view is a higher-level abstraction of the class-interaction
view, where classes are grouped based on the module in which they are instantiated. Modules may also represent compiled binary components such dynamically linked libraries (DLL).

6. **Thread-interaction view.** The thread-interaction view is derived from the object-interactions, and it highlights the threads that execute concurrently and share objects. The view pinpoints objects that can be examined for potential concurrency issues such as race conditions or deadlock.

7. **Feature-interaction view.** This view illustrates the interactions between program features. Features are defined by users as marked-traces. A marked-trace is established manually, during the execution of the program, by specifying the start and end of the trace. Feature interactions are derived from object-interactions automatically.

8. **Feature implementation view.** This view is a mapping between program features and the classes that implement these features.

9. **Feature similarity view.** This view highlights the similarities and dissimilarities between features by matching the call-graphs of features.

10. **Class protocol view.** This view is derived by processing all method invocation sequences of a class to approximate the class usage scenarios as a finite set of regular expressions, where each regular expression describes a usage scenario of the class.

### 4.1 Call-graph View

A call-graph is a directed graph $G(V, E)$ that represents the call structure of a program. It shows the caller-callee relationships. Nodes represent functions (or methods) and edges represent calls. The call-graph is a directed graph. There are two types of call-graphs:

1. **Static call-graphs:** These graphs are extracted from the analysis of the source code.

2. **Dynamic call-graphs:** These graphs are extracted from the runtime data (method invocation events) collected during the execution of the program.
The key difference between the static and dynamic call-graphs is that all edges in a static call-graph are present in the source code, and the call-graph of the program only changes if the source code changes while, in a dynamic call-graph, edges represent static call relationships (relations present in the source code) and dynamic call relationships that only exist when the program is executed due to the use of interfaces (that may have different implementations) or dynamic binding.

To demonstrate these differences, a small program that implements two sorting algorithms is used. The default sorting algorithm is QuickSort, the second is BubbleSort. The user can specify which algorithm to use from the command-line. Each of the algorithms is implemented as a class that implements the ISortAlgorithm interface.

Figures 4.3 and 4.4 show the dynamic call-graphs of the program during two different runs, one with QuickSort and another with BubbleSort as the sorting algorithm. The key observation here is that the same program produces two different dynamic call-graphs. The dynamic call-graph of statically compiled/linked program is always a sub-graph of the static call-graph of the program. However, in a dynamically compiled/linked program, the dynamic-call graph is not necessarily a subgraph of the static call-graph.

**Figure 4.2: Static call graph for the sorting example**

4.2 Object-interaction View

The object-interaction view shows the creation and method invocation relationships between objects. It provides information about the collaboration between objects based on the \textit{depends} and \textit{shares} relations described below. The view is very detailed and not suitable for
program comprehension. However, it serves as the base view for deriving higher level views such as class-interaction and feature-interaction views. An object represents the runtime object reference which resulted from a class instantiation. To create an object-interaction view the profiler must collect object events (create and destroy) and associate method invocations with a pair of corresponding object references.

The depends and shares relationships for objects $O_k$ and $O_j$ are defined as follow:

$$\text{depends}(O_k, O_j) = I(O_k) \cap E(O_j) \neq \phi, \ (j \neq k)$$

$$\text{shares}(O_k, O_j) = I(O_k) \cap I(O_j) \neq \phi, \ (j \neq k)$$

Where,

$$\text{Import}(O_k) = \text{Set of objects used by object } O_k \text{ and created by object } O_j, \ j \neq k$$

$$\text{Export}(O_k) = \text{Set of objects created by object } O_k \text{ and used by object } O_j, \ j \neq k$$

Figure 4.5 shows the object-interaction view of the KWIC program (an implementation of keyword in context algorithm [57]). In the graph a node represents an object, and an edge represents a uses relationship. The arrows with circled-tail indicate that the object attached to the tail created the object attached to the head of the arrow. Object-interaction graphs are typically large and unmanageable for most systems. For this small example, which only created 36 objects, the graph can fit on a single page and is readable. For larger examples, object-interaction views are large and cumbersome, and are only useful to derive higher level views. For example, the view in Figure 4.5 can be abstracted to the class-interaction view shown in Figure 4.6. The list of objects within each class in the graph by default are suppressed (not visible), but are shown in the graph for illustration.

### 4.3 Class-interaction View

The class-interaction view is an abstraction of the object-interaction view, where sets of objects are grouped into classes. Two types of class-interaction views can be derived from the object-interaction view.
Figure 4.5: KWIC: Object-interaction view
Figure 4.6: KWIC: Class-interaction view (with objects shown)

The first type of class-interaction view is derived from object creation and object usage relationships. An example of this type is shown in Figure 4.7. This is the same type of view as the one shown in Figure 4.6, except that the object instances are not shown. In Figure 4.7, there are two types of relationships: creates (represented as an edge with dotted-tail) and uses (represented as a simple edge) relationships derived from the depends and shares relations of object-interactions view. The creates relationship between two classes $C_1$ and $C_2$ implies that an instance of $C_1$ created, and possibly used, an object instance of type $C_2$. The uses relationship between $C_1$ and $C_2$ implies that an instance of $C_1$ used (e.g., invoked a method of) an instance of $C_2$ that was not created by $C_1$. This view is essentially a hybrid static/dynamic view as the entities are static (e.g., classes) and the relationships are dynamic (e.g., invokes and creates). It is far less detailed than the object-interaction view and can be used by developers for program comprehension. The class-interaction view is similar to the inter-class call matrix and histogram of instances grid developed by Pauw et al. [58].
The second type of class-interaction view is derived from the method invocation relationships. An example of this type is shown in Figure 4.8. The nodes in the graph represent classes. The edges in the graph represent method invocation relationships between instances of classes. This type of class-interaction view can be derived from either static or dynamic call-graphs.

The key difference between the two types of class-interaction views is that the view created from object-interactions can only be constructed from runtime events that were collected from the execution of the program. The view shows how objects are created and used by the classes of the program, while, the view created from the call relationships can be constructed from either static or dynamic data, and it shows the invocation relationships between classes. It worth noting that this view may not be complete if it is constructed from static data, because static data may not reveal relationships that only exist at runtime. This case occurs when the program dynamically loads classes and binary components at runtime. Our analysis of Mozilla, which involved a partial execution of the program, uncovered 119,571 unique invocation relationships between class methods, while the source code analysis of the entire source code distribution of Mozilla uncovered only 77,224 relationships between class methods.
Figure 4.8: KWIC: Class-interaction view (based on method invocation)
4.4 Module-interaction View

In this context, a module refers to a binary component such as a dynamically-linked library (DLL) or an executable file (EXE). In a module-interaction view, nodes represent modules and edges represent the interaction between modules. These interactions are invocations between functions/methods of the classes implemented in each module. This view provides a higher-level abstraction of the class-interaction view. Each node of the view encapsulates the class-interaction view of the classes contained in the module.

Figure 4.9 illustrates the module-interaction view for imgjpeg.dll, the JPEG graphics library of Mozilla. This example shows the direct interactions of imgjpeg.dll with other modules. The label header of each node is the name of the module, and the annotations inside the node represent classes and functions that constitute the module.

4.5 Thread-interaction View

The thread-interaction view can be constructed from the object-interaction view by grouping runtime objects according to the thread where the objects were created and used. The thread-interaction view highlights threads that execute concurrently and share objects. The view isolates objects that can be examined for potential concurrency issues such as race conditions or deadlock. Figure 4.10 shows a thread-interaction view derived from the analysis of the Jetty [49] web server. The nodes in the view represent threads, and the edges represent the classes whose instances are shared between threads.

4.6 Remote-interaction View

A key requirement for analyzing distributed systems is the identification of remote interactions between components. The profiler must be able to distinguish between local interactions and remote ones. Remote interactions occur when a component invokes special functions that result in a request/reply messaging protocol between the two distributed components. For example, in Java, the special functions include the read() and write() methods of the java.net.SocketInputStream and java.net.SocketOutputStream classes. In
Figure 4.9: Module-interaction view for imgjpeg.dll module
Figure 4.10: Thread-interaction view

C/C++, the special functions include the recv() and send() functions of the winsock.dll library.

As described in Section 3.1.1, to support the analysis of distributed systems, the data collection and repository subsystems model the concept of communication endpoint entities, and the connects relationship between endpoints.

Figure 4.11 shows the remote interactions of the search-by-author feature of the TechReport program. TechReport is a distributed publication database built in Java using Java Enterprise JavaBeans (EJB/J2EE). The analysis of TechReport is described in detail in Section 5.3. In the diagram, endpoints are represented as diamonds with a local-host:port ⇔ remote-host:port label.

The clusters (subgraphs) in Figure 4.11 show the class-interaction view of the TechReport client (techreport-client) and the TechReport server (techreport-server) respectively. The rectangular nodes represent classes, and diamond-shape nodes represent endpoints. The edges between class nodes represent invocation relationship between instances of the classes, and the edges between nodes endpoint represents connects relationships.
Figure 4.11: Remote-interaction view for the search-by-author feature of TechReport.
4.7 Feature-interaction View

The feature-interaction view captures relationships between features. This view is derived from the object-interaction view by grouping objects based on features that created or used the objects. This view requires that the profiler is able to collect data about runtime objects (creation and usage events), if data about runtime objects is not collected the feature-interaction view cannot be constructed. Features in our analysis are identified in terms of marked-traces, which the user specifies during the execution of a program using a utility to mark the start or the end of each execution trace.

The creation of a feature-interaction view requires identifying various sets of runtime objects. These sets are defined as follows:

\[
\text{Import}(F_k) = \text{Set of objects used (method invocations) by feature } F_k \text{ and created by feature } F_j, j \neq k
\]

\[
\text{Export}(F_k) = \text{Set of objects created by feature } F_k \text{ and used by feature } F_j, j \neq k
\]

These sets help to define the following two relations, which are used to describe how the feature-interaction is constructed:

\[
\text{depends}(F_k, F_j) = \text{I}(F_k) \cap \text{E}(F_j) \neq \phi, \ (j \neq k)
\]

\[
\text{shares}(F_k, F_j) = \text{I}(F_k) \cap \text{I}(F_j) \neq \phi, \ (j \neq k)
\]

In the \text{depends}(F_k, F_j) relation, feature \( F_k \) used objects that were created during the execution of feature \( F_j \). While in the \text{shares}(F_k, F_j) relation, both the \( F_k \) and \( F_j \) features used the same set of objects that were neither created by the \( F_k \) nor by the \( F_j \) feature.

Figure 4.12 shows a features-to-objects grid. The dark rectangles in the diagram are objects that were created during the execution of a feature; light rectangles are objects that were used during the execution of a feature but created by another feature. The content of Figure 4.12 is described as follows:

Feature \( F_1 \) uses objects \( \{O_1, O_2, O_3\} \),

Feature \( F_2 \) uses objects \( \{O_2, O_4, O_5\} \),
Figure 4.12: Feature-interaction grid

Figure 4.13: Feature-interaction and implementation views
Feature $\mathcal{F}_3$ uses objects $\{O_1, O_4, O_6\}$,

Feature $\mathcal{F}_4$ uses objects $\{O_4, O_7\}$,

Class $C_1$ instantiates objects $\{O_1, O_2\}$,

Class $C_2$ instantiates objects $\{O_3, O_4, O_5, O_6\}$,

Class $C_3$ instantiates object $\{O_7\}$

$O_1$ is created during the execution of $\mathcal{F}_1$ and is used by $\mathcal{F}_3$. $O_2$ is created during the execution of $\mathcal{F}_1$ and is used by $\mathcal{F}_2$. $O_4$ is created during the execution of $\mathcal{F}_2$ and is used by $\mathcal{F}_3$ and $\mathcal{F}_4$. The feature-interaction diagram, in Figure 4.13(a), is constructed as follows:

1. Identify the $\text{Import}(\mathcal{F})$ and $\text{Export}(\mathcal{F})$ sets for each feature:

$$\text{E}(\mathcal{F}_1) = \{O_1, O_2\}$$

$$\text{I}(\mathcal{F}_1) = \emptyset$$

$$\text{E}(\mathcal{F}_2) = \{O_4\}$$

$$\text{I}(\mathcal{F}_2) = \{O_2\}$$

$$\text{E}(\mathcal{F}_3) = \emptyset$$

$$\text{I}(\mathcal{F}_3) = \{O_1, O_4\}$$

$$\text{E}(\mathcal{F}_4) = \emptyset$$

$$\text{I}(\mathcal{F}_4) = \{O_4\}$$

2. Identify the $\text{depends}$ and $\text{shares}$ relations between each pair of features using the $\text{E}$ and $\text{I}$ sets:

$$\text{depends}(\mathcal{F}_2, \mathcal{F}_1) = \text{E}(\mathcal{F}_1) \cap \text{I}(\mathcal{F}_2) = \{O_2\}$$

$$\text{depends}(\mathcal{F}_3, \mathcal{F}_1) = \text{E}(\mathcal{F}_1) \cap \text{I}(\mathcal{F}_3) = \{O_1\}$$

$$\text{depends}(\mathcal{F}_3, \mathcal{F}_2) = \text{E}(\mathcal{F}_2) \cap \text{I}(\mathcal{F}_3) = \{O_4\}$$

$$\text{depends}(\mathcal{F}_4, \mathcal{F}_2) = \text{E}(\mathcal{F}_2) \cap \text{I}(\mathcal{F}_4) = \{O_4\}$$

$$\text{shares}(\mathcal{F}_3, \mathcal{F}_4) = \text{I}(\mathcal{F}_3) \cap \text{I}(\mathcal{F}_4) = \{O_4\}$$
3. Draw an edge between $F_j$ and $F_k$ if the $depends(F_j, F_k)$ set is not empty. The $depends$ edge is represented as a solid-arrow, where the arrow head points to the feature that created the object(s) participating in the $depends$ relationship.

4. Draw an edge between $F_j$ and $F_k$ if the $shares(F_j, F_k)$ set is not empty. The $shares$ edge is represented as a dotted-line.

4.8 Feature-implementation View

The data combined from the class-interaction and the trace annotations (marked-traces) is used to produced the feature-implementation view (Figure 4.13(b)), which identifies the classes that are used to implement a given feature. This view provides a simple mapping between features and implementation-level classes. This view is not unique to our work; several research efforts [12, 20, 79, 80] have identified this kind of mapping. The view is represented as a graph, where the nodes represent features and classes, and edges represent instantiation relations between features and classes.

4.9 Feature-similarity View

The feature similarity view is represented as a matrix, and is computed by approximately matching the call-graph(s) of each pair of features. The similarity measure helps the engineer identify similar features and, thus, guides him/her to learn about the implementation of a feature, by studying similar features. The similarity measure also helps the engineer to assess the impact of a change of one feature on other features in the software system.

The similarity is measured by approximately matching the graphs that represent the call-graphs of two features $F_1$ and $F_2$ with associated call-graphs $G_1(V_1, E_1)$ and $G_2(V_1, E_2)$ respectively. The similarity between $F_1$ and $F_2$ is computed as:

$$Similarity(F_1, F_2) = \frac{|E_1 \cap E_2|}{|E_1 \cup E_2|}$$

(4.1)

Where, $V_k$ is the set of node in $G_k$, which is the set methods invoked during the execution of feature $F_k$. $E_k$ is the set of edges in $G_k$, which is the set of call relationships in the
call-graph of feature $F_k$. 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.

The somewhat cruder similarity approximation can be computed by matching the sets of nodes of the call-graphs, as follows:

$$\text{Similarity}(F_1, F_2) = \frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}$$

Both measurements produce results of similar quality. The above two measures are based on the set similarity measure, which is referred to as the Jaccard Index. The set-based similarity is an acceptable approximation for matching call-graphs and has been used in previous work by other researchers [11, 10, 77].

4.10 Class-usage Protocol View

This section describes a software view that captures class usage scenarios from method invocations made to objects at runtime. Knowing the sequence by which the methods of a class can be invoked is important to software engineers who need to understand how to use the class. An algorithmic approach is used to create class usage protocol views, which involves computing the distance between every pair of method invocation sequences. The algorithm then employs the notion of canonical sets [18, 17] to categorize the method sequences into groups of similar sequences, where each group represents a usage scenario for a given class.

For example, assume that the software component under scrutiny is a class called File. This class has an interface that consists of the methods $\text{open}()$, $\text{read}()$, $\text{write}()$, $\text{close}()$, and the constructor method $\text{File}()$. In trying to understand how the $\text{File}$ class works, one of the questions that needs to be answered is: what is a valid sequence of method invocations? Is the sequence $<\text{open}(), \text{read}(), \text{close}()>$ a valid sequence, or should a $\text{write}()$ be performed following a $\text{read}()$?

Specific method invocation sequences often describe different class usage scenarios (i.e., how a class is used by other classes). For example, for the $\text{File}$ class we have the following method invocation sequences and corresponding usage scenarios:
- The sequence \(<\text{open}, \text{close}\>\), which can be used to check the existence of a file.

- The sequence \(<\text{open}, \text{read}+, \text{close}\>\), which can be used to read the contents of a file. (+ indicates one or more repetitions.)

- The sequence \(<\text{open}, \text{write}+, \text{close}\>\), which can be used to write the contents of a file.

- The sequence \(<\text{open}, \text{read}+, \text{write}+, \text{close}\>\), which can be used to read a file, manipulate its content (e.g., sorting) and, finally, store the manipulated content back into the file.

Of course, these are just some of the many usage scenarios for the \texttt{File} class.

The intuition for our approach is based on the observation that method invocation sequences for a particular class usage scenario share certain similarities, but are fairly dissimilar to the sequences that correspond to other usage scenarios. For example, considering the method invocation sequences and usage scenarios for the \texttt{File} class that were described above, we observe that all of the sequences used for just reading the contents of a file are of the form \(<\text{open}, \text{read}+, \text{close}\>\), while the sequences used for reading, manipulating, and storing the contents of a file are of the form \(<\text{open}, \text{read}+, \text{write}+, \text{close}\>\).

The identification of class usage scenarios from sequences of method invocations involves: (a) collecting runtime information about method invocations made to profiled object instances of a class, (b) analyzing this information to create symbolic expressions representing the method invocation sequences for the given class, (c) identifying a small subset of sequences that best characterizes all method invocation sequences, and (d) using the elements of this small subset to identify groups of sequences that represent various class usage scenarios.

The process of computing the class Message Sequence Protocol (MSP) involves computing the distance between every pair of method invocation sequences using a distance measure (i.e., edit distance of ordered sequences). Using this distance measure, the algorithm computes the canonical set of method sequences, which is a small subset of the sequences that
best characterizes the elements of the original set. By definition, each element of the canonical set is associated with a group of method sequences, called the *canonical group*, which correspond to a class usage scenario. In our work, we used the algorithm developed by researchers at Drexel University [18, 17], for finding the canonical set of the set of method sequences.

There exists a cost associated with each valid operation. The cost represents the amount of a single modification required to transform one sequence to another. The cost is not necessarily a uniform function for different kinds of operations. For example, inserting a method call that does not already exist in the sequence is a more costly operation than inserting a call that already exists. The cost is a function of the frequency of a particular method call in the sequence. For example, inserting a method call represented by the symbol $R$ into the sequence $<ORRRC>$ will cost more than inserting it into the sequence $<Orrrrrrrrc>$. Furthermore, the cost is a function of the location. For example, inserting an $R$ after the first $R$ in the sequence $<ORRWWC>$ is cheaper than inserting the $R$ after the first $W$. The cost function, which accounts for both the frequency distribution of calls in a sequence and their location, is a windowed inverse exponential function. A closed form description of the cost function can be found elsewhere [4].

Figure 4.14 illustrates the high-level architecture of the analysis subsystem of the class usage protocol recovery. The *analysis* subsystem creates an object-interaction model from the runtime information, computes the canonical set, and identifies the various class usage scenarios. It consists of three components: MS Extractor, MS Classifier, and RE Approximator.

The *MS Extractor* (Method Sequence Extractor) creates the object-interaction model and extracts the method invocation sequences for a given class. Objects are uniquely identified by their runtime references, and thus each object has its own method sequence. The MS Extractor then constructs symbolic representations of the method invocation sequences, which are used by the MS Classifier to compute the canonical set.

The *MS Classifier* (Method Sequence Classifier) is the component that implements the algorithm for computing the canonical set, and uses the resulting elements to create the
Figure 4.14: Analysis subsystem: Class-usage estimation
canonical groups of method invocation sequences that correspond to class usage scenarios. The MS Classifier is implemented using MatLab and Perl scripts. The algorithms for computing the canonical sets and groups are described in detail elsewhere [18, 17].

The *RE Approximator* (Regular Expression Approximator) is an optional component whose objective is to encode the elements of the canonical groups, and hence the sequences that define the class usage scenarios, into an expression that improves readability for the end user. The approximation process is described below in more detail.

Next, we describe how the elements of the canonical set are used to group the method invocations sequences.

**Computing canonical groups**

Given $P = \{p_1, ..., p_n\}$, the set of method invocation sequences, and $P' = \{q_1, ..., q_k\}$, the set of canonical sequences, the MS Classifier uses a minimum distance classification approach to compute the canonical groups. The steps used to compute the canonical groups are as follows:

1. For each sequence, $p_i$, in $P$, compute its similarity to all of the canonical sequences in $P'$.
2. Place the sequence, $p_i$, in a group that corresponds to the canonical sequence, $q_i$, to which it is most similar (i.e., smallest edit distance measure.)
3. In the event of a tie in max similarity, place $p_i$ in all of the corresponding canonical groups.

Each resulting canonical group contains similar sequences that represent a usage scenario of the class.

**Regular expression approximation**

The purpose of this approximation is to simplify the groups of the canonical set. Our implementation is based on work in the areas of string matching [44, 5] and the approximation of regular expressions [8]. The outline of the approximation process is as follows:
1. Compute the longest common subsequence (LCS) of each method sequence. The LCS sets of all sequences are combined into one LCS set, which is used in the next step to ensure that the sequence segmentation of all sequences is performed in the same order for all sequences.

2. Compress the sequences using the combined LCS set. This step segments each sequence into subsequences and finds the repetitions of each subsequence. Note that each subsequence is a member of the LCS set. For example, the following two sequences:

\[
\text{ABIKLMHCDKEFGH} \cdots \text{EFGH} \\
\text{ACDEFGHEFGHEFGH} \cdots \text{EFGHJ}
\]

Can be compressed into:

\[
(A)^1(BI)^1(K)^1(LM)^1(H)^1(CD)^1(K)^1(EFGH)^{45} \\
(A)^1(CD)^1(EFGH)^{13} (J)^1
\]

3. Align the compressed sequences to produce a single expression that represents all of the sequences. The alignment is performed using the anchor subsequences, which are the common repeated subsequences. Empty (ε) subsequences are inserted as needed to produce the aligned sequences. For the above example, the alignment process produces the alignment shown in the first two rows of Table 4.1. Then the aligned sequences are combined into a single expression as shown in the third row of Table 4.1.

4. Approximate the aligned single expression as a regular expression using the following rules:

\[
\begin{array}{ccccccccccccc}
(A)^1 & (BI)^1 & (K)^1 & (LM)^1 & (H)^1 & (CD)^1 & (K)^1 & (EFGH)^{45} & \epsilon \\
(A)^1 & (CD)^1 & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon & (EFGH)^{13} & (J)^1 \\
(A)^1 & (BI|CD)^1 & (K)^{0,1} & (LM)^{0,1} & (H)^{0,1} & (CD)^{0,1} & (K)^{0,1} & (EFGH)^{13,45} & (J)^{0,1} \\
(A) & (BI|CD)^1 & (K)^{7} & (LM)^{7} & (H)^{7} & (CD)^{7} & (K)^{7} & (EFGH)^{+} & (J)^{7}
\end{array}
\]

Table 4.1: Sequence segment alignment
• If the subsequence has a minimum periodicity of zero and a maximum periodicity of one, then the subsequence is optional \(?\) (zero or one).

• If the subsequence has a monotonically increasing periodicity starting at zero, then set the periodicity to \(*\) (zero or more).

• If the subsequence has a monotonically increasing periodicity starting at one, then set the periodicity to \(+\) (one or more).

The approximated regular expression is shown in the fourth row of Table 4.1.

The approximated regular expressions of the method sequences represent the usage scenarios of the class under study.

4.11 Summary

This chapter describes the set of software views developed as part of our research. The views include feature-interaction, feature-implementation, feature-similarity, remote-interaction, thread-interaction, structural, protocol approximation views.

Our key contributions include the feature-interaction, remote-interaction, thread-interaction, and class protocol views. The feature-interaction view is useful to assess the change impact of a feature on other features, as well as it can help determined the dominant feature(s) of an application. The remote-interaction view helps identify remote invocations between component of a distributed system. This view is unique to our research in identifying remote interactions through dynamic analysis. The thread-interaction view pinpoints to shared classes between threads that may cause deadlock or race conditions for multi-threaded applications. Further work is needed in this area to allow the automatic identification of potential deadlock or race conditions between threads. The class protocol view a protocol that describes the order in which the methods are invoked, as well as the set of scenarios that the class is used. Our contribution is attempting to find an compact approximation of the set of method sequences of a class and presenting them as regular expressions. Further work on this view includes the study of groups of collaborated classes that can be considered as a subsystem, and approximate their protocol as a subsystem usage protocol. This is a
harder problem, but it will provide a more useful information to an engineer, because in
genral to accomplish a task more than one class is used.
Chapter 5: Case Study

This chapter describes a case study that demonstrates the effectiveness our software comprehension environment. The software systems involved in the case study are outlined in Table 5.1.

Table 5.1: Systems analyzed

<table>
<thead>
<tr>
<th>Systems</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla: Web Browser</td>
<td>Win32/C,C++</td>
</tr>
<tr>
<td>Jext: Text Editor</td>
<td>Java</td>
</tr>
<tr>
<td>TechReport: Publications database</td>
<td>Java/EJB</td>
</tr>
</tbody>
</table>

Figure 5.1 outlines the program analysis process for each system in the case study. The four phases are common to all three systems analyzed; the differences are in the details of the first two phases. The build and profiling phases depend on the data collection tools used. The data import and analysis phases are the same for all three systems in the case study. The build phase is performed on the program under study to prepare it for profiling. This activity may involve re-compiling the program with instrumentation (as in the case of C/C++ programs) or doing nothing (as in the case of Java programs). The steps of profiling and analyzing a software system are outlined below:

- **Run the program in profiling mode:** When the profiler is loaded it runs the Trace-Marker wizard and prompts the user for a file name in which to store the runtime events. **WinProf** is used to profile Win32 programs and **JvProf** is used to
profile Java programs. When profiling a distributed application, such as TechReport, the logical time server must be started. The details of this step depends on the application to be profiled. The tasks performed to profile each application in the case study are described Sections 5.1.1, 5.2.1 and 5.3.1.

- **Load the runtime events into the repository using the dbimport tool:** This tool prompts the user for the XML file containing the runtime events, a project name, programming language, and the JDBC connection parameters to the repository. Figure 5.2 shows the dbimport tool.

![Figure 5.2: dbimport tool](image)

- **Load the main tool SceTool:** Figure 5.3 shows the main window of the software comprehension environment, which provides capabilities to query the repository in SMQL language. In the main window, there are four areas: the database tables and views of the repository (upper-left corner), the programming languages models stored in the repository (lower-left corner), the SMQL text editor (upper-right corner), and the SMQL query results (lower-right corner).

- **Open an SMQL editor window to load (or write) the SMQL code to be
Figure 5.3: SceTool: Environment main window
executed.

Table 5.2 outlines the software views that were created for each system of the case study. The types of software views that can be created depends on the program artifacts that can be collected from the profiler. Different profilers have different capabilities. For example, the Win32 profiler used to analyze Mozilla does not collect data about runtime objects (just data about function and method calls), and therefore, an object-interaction view cannot be created for Mozilla while the Java profiler, used to analyze the Jext and TechReport systems, can collect data about runtime objects, and thus produce object-interaction views for these systems.

Each system of the case study demonstrates different aspects of our research. The analysis of the Mozilla system demonstrates the effectiveness of the tools in profiling and studying large scale systems written in C/C++ on the Win32 platform. Feature, module, and class views based on call-graphs are created during the analysis of Mozilla. The Jext analysis demonstrates the effectiveness of our tools in profiling and studying of medium-sized Java applications. Feature and class views based on object-interactions are created for Jext. The analysis of TechReport demonstrates the effectiveness of our tools in profiling and studying of distributed systems. Feature, class, remote interaction, and thread interaction views based on object-interactions are created for TechReport.

Table 5.2: Outline of software views produced for systems of the case study

<table>
<thead>
<tr>
<th></th>
<th>Mozilla</th>
<th>Jext</th>
<th>TechReport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-interaction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Class-interaction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Module-interaction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Feature-implementation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Feature-interaction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Feature-similarity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Remote-Interaction</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Thread-Interaction</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Class-usage protocol</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.1 Mozilla: Web Browser

Mozilla is an open-source web browser ported to almost every operating system platform. Mozilla is second to Microsoft’s Internet Explorer, with 8% of the market share of Internet browsers [51]. In addition to the web browser as its core functionality, Mozilla includes other Internet 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 implemented primarily in C++. Smaller part of the program is written in C. The Mozilla version we analyzed (Version 1.0.1) includes over 4.4 MLOC located in close to 13,000 source files in about 1,200 directories. 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 installation programs.

The features of the case study focus mainly on the web browser and partially on the e-mail features of Mozilla. Table 5.3 outlines Mozilla’s size. Clearly, the thousands of classes and relationships complicate program understanding and maintenance.

<table>
<thead>
<tr>
<th>Table 5.3: Mozilla files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Header files</td>
</tr>
<tr>
<td>C files</td>
</tr>
<tr>
<td>C++ files</td>
</tr>
<tr>
<td>IDL files</td>
</tr>
<tr>
<td>C++, C &amp; header files</td>
</tr>
</tbody>
</table>

Table 5.4 summarizes the code coverage statistics that were recovered by the dynamic analysis of Mozilla. In the table, modules correspond to dynamically-linked libraries and the main executable file of Mozilla. 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.

Table 5.4: Mozilla code coverage

<table>
<thead>
<tr>
<th></th>
<th>Methods</th>
<th>Classes</th>
<th>Modules</th>
<th>XPCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executed</td>
<td>30789</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loaded</td>
<td>47892</td>
<td>4614</td>
<td>61</td>
<td>694</td>
</tr>
<tr>
<td>Total</td>
<td>77842</td>
<td>7875</td>
<td>102</td>
<td>1089</td>
</tr>
</tbody>
</table>

As stated earlier, 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 our analysis, which involves the execution of a small subset of Mozilla’s features, the dynamic analysis found 119,571 unique invocation relationships between class methods, while the source code analysis, performed using SourceNavigator [67], 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 DLL. The additional 42,347 relationships discovered via dynamic analysis were interactions between classes in various binary modules. A specific example is the nsObserveService class, which implements the nsIObserverService interface. The dynamic analysis found 22 distinct relationships between nsObserveService and other classes in 11 binary modules. These relationships would not have been discovered using static analysis.

It is worth noting that our analysis of Mozilla differs from the source code-based analysis of Mozilla by Godfrey et al. [27], because our work is a feature-driven and relies on dynamic rather than a static analysis.

5.1.1 Mozilla: Build and Profiling

The profiling is performed using code instrumentation that inserts function handlers to intercept function-entry and exit events. The function-entry handler is implemented using 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, which is 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 function is invoked, and the number of exceptions thrown by each function. Upon the termination of the program’s execution, the collected data is stored in flat files, which are subsequently imported into the data repository for further analysis.

Figure 5.4 illustrates a refinement of the software comprehension environment architecture as it pertains to the analysis of Mozilla.

![Tool Architecture for the analysis of Mozilla](image)

**Figure 5.4: Tool Architecture for the analysis of Mozilla**

To perform the instrumentation, one of Mozilla’s configuration files (`./config/WIN32`) was changed by adding the following lines of code:

```bash
## To enable Function-entry hook by the VS compiler.
!if defined(MOZ_GH_HOOK)
OS_CFLAGS=$(OS_CFLAGS) -Gh
OS_LIBS=$(OS_LIBS) $(MOZ_GH_HOOK)
!endif
```

The purpose of these lines of code is to verify whether the `MOZ_GH_HOOK` environment variable is defined. If the variable is defined, then the `/Gh` compiler option is added to the
compiler command, and the function hook library is added to the common libraries required to build Mozilla. The `MOZ_GH_HOOK` variable holds the full path of the function hook library. This library defines three functions:

- **.penter** function, which is required when compiling with the `/Gh` option. The `/Gh` option directs the compiler to insert a call to the `penter` function at the beginning of every function in the source code. The function takes no arguments, and the addresses of the called and caller functions are read from the stack frame of the called function. When the `penter` function is invoked it performs the following:
  - if the function is invoked for the first time, it invokes the `initialize` function (described below);
  - reads the memory addresses of the called and caller functions from the stack frame;
  - modifies the return address of the invoked function to be the `pexit` function, so that the function-exit event can be captured.

- **.pexit** function, which is invoked when a function exits. After the `pexit` function is invoked, the original return address of the profiled function is restored.

- **initialize** function, which is invoked when `penter` is invoked for the first time. This `initialize` function performs the following:
  - initializes a semaphore used by the `penter` and `pexit` functions to avoid race conditions when profiling multi-threaded programs, because `penter` and `pexit` are shared by all running threads;
  - loads the custom (pluggable) data collection libraries. In our case, there are two dynamically-linked libraries: `PEnterHandler.dll` and `TraceMarker.dll`. `PEnterHandler.dll` provides two functions `gh_functionEntry` and `gh_functionExit`, which are invoked from `penter` and `pexit`, respectively. `TraceMarker.dll` provides a simple wizard that enables users to specify the start and end of program use-cases or features.
For the build phase, the environment variable `MOZ_GH_HOOK` is set to the full path of the `penter` library. For example,

```
set MOZ_GH_HOOK=D:\Distribution\sce\ext\penter.lib
```

For the profiling phase, the following environment variables are set as follows:

```
set GH_PROFILE_ENABLED=1
set GH_HANDLER_DLL=D:\Distribution\sce\ext\PEnterHandler.dll
set TRACE_MARKER_LIBRARY=D:\Distribution\sce\ext\TraceMarker.dll
set GH_OUTPUT_PATH=D:\Distribution\sce\data
```

The `GH_PROFILE_ENABLED` variable enables (value = 1) or disables (value ≠ 1) profiling, when running Mozilla. The `GH_HANDLER_DLL` and `TRACE_MARKER_LIBRARY` variables specify the full path of the data collection libraries. The `GH_OUTPUT_PATH` variable specifies the location where the collected data is stored, the data is then imported into the repository using the `dbimport` tool. (Note that D:\Distribution\sce\ is the tool’s installation directory).

### 5.1.2 Mozilla: Features

We identified an initial set of Mozilla features (or use-cases) that are characteristic of any web browser’s functionality. Tables 5.5(a) and 5.5(b) outline the nine features of the case study (for C++ and C code) and their coverage statistics: number of loaded modules, number of instantiated classes, number of invoked methods, and the number of method-entry events created during the execution of each feature.

Next, we describe the similarity between each pair of the Mozilla features. The feature similarity matrix is computed from the caller-callee relationships of the methods invoked while executing each feature. The similarity measure helps the engineer identify similar features and, thus, guides him/her to learn about the implementation of a feature 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. In our analysis of Mozilla, the similarities between some features are obvious, for example, the strong similarity between the `open-url` and `bookmark-open` features.
Table 5.5: Features coverage statistics

(a) Runtime statistics (C++ code)

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

(b) Runtime statistics (C code)

<table>
<thead>
<tr>
<th>Features</th>
<th>Modules</th>
<th>Functions</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>save-page</td>
<td>38</td>
<td>859</td>
<td>2,583,686</td>
</tr>
<tr>
<td>print-page</td>
<td>34</td>
<td>848</td>
<td>2,026,634</td>
</tr>
<tr>
<td>open-url</td>
<td>39</td>
<td>791</td>
<td>4,418,660</td>
</tr>
<tr>
<td>bookmark-add</td>
<td>23</td>
<td>413</td>
<td>702,545</td>
</tr>
<tr>
<td>open-link</td>
<td>37</td>
<td>756</td>
<td>1,947,177</td>
</tr>
<tr>
<td>startup</td>
<td>46</td>
<td>1,027</td>
<td>9,103,695</td>
</tr>
<tr>
<td>bookmark-open</td>
<td>35</td>
<td>724</td>
<td>1,955,112</td>
</tr>
<tr>
<td>shutdown</td>
<td>42</td>
<td>514</td>
<td>6,540,627</td>
</tr>
<tr>
<td>send-page</td>
<td>50</td>
<td>1060</td>
<td>11,631,952</td>
</tr>
<tr>
<td>unmarked traces</td>
<td>38</td>
<td>802</td>
<td>4,204,130</td>
</tr>
</tbody>
</table>
The feature similarity matrix, shown in Table 5.6, is computed using the Jaccard index similarity function, which is defined as:

$$ Similarity(F_1, F_2) = \frac{|E_1 \cap E_2|}{|E_1 \cup E_2|} $$

Where, $E_k$ is the set of caller-callee relationships of the methods invoked while executing feature $k$, $|E_1 \cap E_2|$ is the cardinality of the intersection of $E_1$ and $E_2$, and $|E_1 \cup E_2|$ is the cardinality of the union of $E_1$ and $E_2$.

During the analysis, common classes and modules can be either included or filtered out. A common class or module is an entity that is used in the implementation of a high percentage of features. Filtering out such entities not only reduces the clutter of the views, but also emphasizes the uniqueness of each feature. Filtering out common modules also provides a better measure of similarity, for example, the similarity between semantically similar features 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 features decreases significantly when the common classes and modules are filtered out.

### 5.1.3 Mozilla: Structural Views

Next, we describe some of the structural views constructed from the runtime data. These views capture the portions of Mozilla’s architecture that pertain to each feature. These views are constructed as a hierarchy with different levels of abstraction (Figure 5.5). The lowest level represents the caller-callee relationships between methods that are invoked during the execution of the features. 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 feature, or a use-case, refers to an externally visible functionality of a 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
Table 5.6: Feature similarity matrices

(a) Similarity of features with all of the modules and classes

<table>
<thead>
<tr>
<th></th>
<th>save-page</th>
<th>print-page</th>
<th>open-url</th>
<th>bookmark-add</th>
<th>open-link</th>
<th>startup</th>
<th>bookmark-open</th>
<th>shutdown</th>
<th>send-page</th>
</tr>
</thead>
<tbody>
<tr>
<td>save-page</td>
<td>100</td>
<td>69</td>
<td>57</td>
<td>48</td>
<td>57</td>
<td>57</td>
<td>61</td>
<td>44</td>
<td>61</td>
</tr>
<tr>
<td>print-page</td>
<td>100</td>
<td>56</td>
<td>48</td>
<td>59</td>
<td>51</td>
<td>65</td>
<td>44</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>open-url</td>
<td>100</td>
<td>100</td>
<td>42</td>
<td>77</td>
<td>50</td>
<td>76</td>
<td>43</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>bookmark-add</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>43</td>
<td>40</td>
<td>48</td>
<td>42</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>open-link</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>45</td>
<td>84</td>
<td>46</td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>startup</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>47</td>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>bookmark-open</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>shutdown</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

(b) Similarity of features without the common modules

<table>
<thead>
<tr>
<th></th>
<th>save-page</th>
<th>print-page</th>
<th>open-url</th>
<th>bookmark-add</th>
<th>open-link</th>
<th>shutdown</th>
<th>send-page</th>
</tr>
</thead>
<tbody>
<tr>
<td>save-page</td>
<td>100</td>
<td>67</td>
<td>45</td>
<td>12</td>
<td>53</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>print-page</td>
<td>100</td>
<td>35</td>
<td>15</td>
<td>15</td>
<td>45</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>open-url</td>
<td>100</td>
<td>100</td>
<td>10</td>
<td>63</td>
<td>37</td>
<td>17</td>
<td>29</td>
</tr>
<tr>
<td>bookmark-add</td>
<td>100</td>
<td>100</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>open-link</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>36</td>
<td>81</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>startup</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>bookmark-open</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>shutdown</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>send-page</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
feature node produces the module-interaction view of the feature. Selecting more than one node produces a view that contains the selected nodes and the detailed interactions between them. For example selecting two feature nodes produces a view with the two features along with the modules shared between the features as nodes, and the uses relationships between the shared modules as edges.

Figure 5.6 shows the list of Mozilla features that were executed. The module-interaction view for a given feature can be created by selecting the node representing the feature. For example, an engineer can explore the send-page feature by double-clicking on the send-page node, which will construct the module-interaction view of the send-page feature as shown in Figure 5.7. In this view, nodes represent clusters of modules and edges represent the interaction between the modules in the clusters. The label of each cluster denotes the dominant module within the cluster. The module-interaction views can be annotated with simple metrics for each module that are helpful to assess of 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 5.7. 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.

Selecting more than one feature node can produce an implementation view of the selected features, as shown in Figure 5.8. This view shows which modules contribute to the implementation of each selected feature. Note that, in several cases, a module is used in the implementation of more than one feature. This view is useful to engineers who have to modify a module in response to a change in a feature (use-case) because the views expose the features that depend on that module.

Figure 5.9 shows the SMQL code used to analyze the Mozilla features. The SMQL filter `AnalyzeUseCases` analyzes the `allRelations` relation set and produces the feature graphs, and the feature-implementation views. The specific views, such as the view shown in Figure 5.7, are reached interactively through navigation.

Further exploration can be performed at the module level to reveal the class-interaction view of the module as shown Figure 5.10 for the `mime.dll` module. In Figure 5.10, nodes represent classes and edges represent runtime invocations between the member methods of the object instances of the classes. A complementary view of the class-interaction view is the module-interaction view centered around a module of interest (`mime.dll` in this case). This view highlights the modules that the `mime.dll` module directly interacts with, as shown in Figure 5.11. This view is constructed by selecting the module of interest and specifying to
Figure 5.7: Module-interaction view (clustered) for the send-page feature
Figure 5.8: Feature-implementation view
/**
 * Mozilla - Use-cases
 * Computes the use-case views for a Mozilla from the list of modules and
 * traces defined in modulesOfInterest and tracesOfInterest.
 */

// Declare the SMQL filter that analyzes Mozilla's use-cases.
import java:serg.sc.filter.moz.AnalyzeUseCases as AnalyzeUseCases ;

// Define the list of modules of interest
modulesOfInterest = {"*"} ;

// Define the list of traces (features) of interest
tracesOfInterest = {"*"} ;

// Define relationships between the entities contained in "modulesOfInterest"
RelationSet allRelations
{
    // Define the entity-set for the entities contained in modulesOfInterest
    EntitySet EntitiesOfInterest
    {
        type = {"method", "function"} ;
        include (container) = modulesOfInterest ;
    }

    // Include the traces of interest (in our case = ALL)
    include (trace) = tracesOfInterest ;

    // Define the relationships between EntitiesOfInterest and all
    // other entities.
    include ("invoke") = { EntitiesOfInterest -> EntitiesOfInterest } ;
}

// Invoke the SMQL filter to analyze the use-cases
UserCases = AnalyzeUseCases(allRelations) ;

Figure 5.9: SMQL: Mozilla feature analysis
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.

**Figure 5.10: Class-interaction view for the mime.dll module**

**Figure 5.11: Module interaction view of mime.dll module**

The SMQL code used to construct the views in Figures 5.10 and 5.11 is shown in Figure 5.12. The SMQL source code computes the local relationships of a set of binary modules defined in the modulesOfInterest variable, as well as the relationships to and from these modules. In the example, modulesOfInterest, only defines one module (mime.dll).
First, the entities of interest are defined, which are the entities contained in *mime.dll* that are of type function or method. Second, the local relationships are defined in *localRels* as a RelationSet. The relation set is defined in the *include* filter, where the source and target entities belong to the *EntitiesOfInterest* entity set. Third, the relationships to any entity in the *EntitiesOfInterest* set are defined in the *fanInRels* relation set. Fourth, the relationships from any entity in *EntitiesOfInterest* to any entity in the program are defined in the *fanOutRels* relation set. Finally, the union of three relation sets forms the set of all relationships between the members of the *EntitiesOfInterest* entity set. A partial visualization of the result of executing the SMQL code is shown Figure 5.11. The visualization shows the module interactions of the *mime.dll* module with other modules, and Figure 5.10, which shows the class interactions of the *mime.dll* module. A compact version of the SMQL code of the example is shown in Figure 5.13, where all entity sets and all relationship sets are defined inside a RelationSet block.

For large systems like Mozilla, the graphs representing the views are large with hundreds of relationships, which makes them 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) to cluster the graphs of each view.

### 5.2 Jext: Text Editor

Jext [1] is a programmer’s text editor, written in Java, which supports many languages (*e.g.*, C/C++, Java, XSLT, \( \text{T}_{\text{eX}} \), HTML). It provides features such as adding/removing document bookmarks of documents, categorizing opened documents through workspaces, directly opening zipped files, retrieving files from the Internet, mailing source code, and executing system commands within the internal console. The Jext version we analyzed (Version 3.2) includes approximately 98 KLOC located in 565 Java source files. Figure 5.14 shows a screen dump of the Jext text editor.

In our study, we marked traces to identify a subset of the Jext features. The scenarios
/**
 * Mozilla - Example-1
 * Computes the class and module-interactions views for a given modules
 * defined "modulesOfInterest" list of modules.
 */

// Define the list of modules of interest
modulesOfInterest = "mime.dll";

// Define the entity-set for the entities contained in the module of interest.
EntitySet EntitiesOfInterest
{
    type = ["method", "function"];
    include (container) = modulesOfInterest;
}

// define the local relationships between the entities of interest
RelationSet localRels
{
    include ("invoke") = [EntitiesOfInterest -> EntitiesOfInterest];
}

// Define the relationships of all entities references the entities of interest
RelationSet fanInRels
{
    EntitySet I
    {
        type = ["method", "function"];
        include (container) = ["*"];
        exclude (container) = modulesOfInterest;
    }
    include ("invoke") = [I -> EntitiesOfInterest];
}

// Define the relationships of all entities referenced by any entity of interest
RelationSet fanOutRels
{
    EntitySet O
    {
        type = ["method", "function"];
        include (container) = ["*"];
        exclude (container) = modulesOfInterest;
    }
    include ("invoke") = [EntitiesOfInterest -> O];
}

// Now, compute the union of all relationships.
allRelations = localRels + fanInRels + fanOutRels;

Figure 5.12: SMQL: Mozilla class/module interaction example
/*
 * Mozilla - Example-1
 * Computes the class and module-interactions views for a given modules
 * defined "modulesOfInterest" list of modules.
 */

// Define the list of modules of interest
modulesOfInterest = {"mime.dll"};

// define relationships between the entities contained in "modulesOfInterest" and
// all other entities.
RelationSet allRelations
{
  // Define the entity-set for the entities contained in modulesOfInterest
  EntitySet EntitiesOfInterest
  {
    type = {"method", "function"};
    include (container) = modulesOfInterest;
  }

  // Define the entity-set of entities other than EntitiesOfInterest
  EntitySet others
  {
    type = {"method", "function"};
    include (container) = {"*"};
    exclude (container) = modulesOfInterest;
  }

  // Define the relationships between EntitiesOfInterest and all
  // other entities.
  include ("invoke") = { EntitiesOfInterest -> EntitiesOfInterest,
                           others -> EntitiesOfInterest,
                           EntitiesOfInterest -> others};
}

Figure 5.13: SMQL (Rewrite): Mozilla class/module interaction example
Figure 5.14: Jext: Programmer’s text editor
used to mark the selected features are: (a) opening three different documents (a Java source file, an HTML file from the web, and a HTML document from the bookmarks), (b) performing edit activities (copy, cut, paste) on the Java source code documents, (c) searching for a string in the Java source file documents, (d) searching and replacing, (e) adding bookmarks, and (f) e-mailing the Java source code.

Table 5.7 summarizes the coverage statistics of the Jext program. The counts in the table exclude common packages such as java.* and javax.*.

<table>
<thead>
<tr>
<th>Table 5.7: Jext: Coverage statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
</tr>
<tr>
<td>257/479</td>
</tr>
</tbody>
</table>

(1) The number of classes exercised versus the total number of classes in the source code
(2) The number of methods exercised versus the total number of methods in the source code

5.2.1 Jext: Build and Profiling

Re-compilation is not needed for Java applications such as Jext. The Java profiler is loaded as a dynamically-linked library by the Java virtual machine (JVM). The profiling of Jext is performed, simply, by running the following script:

```bash
SET CLASSPATH=..\lib\jython.jar;..\lib\dawn.jar;..\lib\jext.jar
jvprof org.jext.Jext
```

JvProf is the Java profiler, which is a wrapper for invoking Java and loading the Java profiler library.

Figure 5.15 illustrates a refinement of the software comprehension environment architecture as it pertains to the analysis of Jext.

5.2.2 Jext: Features

Table 5.8 shows a summary of the statistics for each feature, which includes the number of classes loaded, the number of methods executed, and the number of events created for each
Figure 5.15: Tool Architecture for the analysis of Jext

feature.

The SMQL code used to derive the software views from the runtime events is listed in Figure 5.16. First, we define two sets of events (EventSet): JextEvents and NetEvents. The JextEvents set includes events that were created during the execution of the marked-traces, but excludes events that were created from the standard packages such as java.* and javax.*. Events that are created during startup and shutdown of the application are excluded.

The second event set is NetEvents, which includes only java.net.* events pertaining to the remote method invocations of a distributed application. The finalize method is excluded to prevent the mis-identification of interactions, since this method is invoked by the garbage collector and the JVM does not guarantee which thread will invoke this method. The union of the two event sets is computed using the “+” operator.

Finally, we invoke the event analyzer (AnalyzeEvents) to create the dynamic views from the input event set. The returned value results is a set of sets, where each element represents a RelationSet that defines a view or a part of a view. Figure 5.19 shows the results returned from the (AnalyzeEvents) analyzer.

Figures 5.17 and 5.18 show the feature-interaction and feature-implementation views
Table 5.8: Jext: Feature coverage statistics

<table>
<thead>
<tr>
<th>Feature</th>
<th>Methods</th>
<th>Classes</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>bookmark-add</td>
<td>63</td>
<td>41</td>
<td>1,840</td>
</tr>
<tr>
<td>bookmark-open-doc</td>
<td>116</td>
<td>58</td>
<td>25,166</td>
</tr>
<tr>
<td>change-lang</td>
<td>45</td>
<td>35</td>
<td>3,044</td>
</tr>
<tr>
<td>edit-copy</td>
<td>30</td>
<td>17</td>
<td>4,638</td>
</tr>
<tr>
<td>edit-cut</td>
<td>39</td>
<td>30</td>
<td>1,958</td>
</tr>
<tr>
<td>edit-paste</td>
<td>66</td>
<td>44</td>
<td>5,354</td>
</tr>
<tr>
<td>email-doc</td>
<td>101</td>
<td>44</td>
<td>8,078</td>
</tr>
<tr>
<td>file-open-doc</td>
<td>149</td>
<td>81</td>
<td>92,686</td>
</tr>
<tr>
<td>search</td>
<td>36</td>
<td>19</td>
<td>8,980</td>
</tr>
<tr>
<td>search-replace</td>
<td>111</td>
<td>48</td>
<td>118,128</td>
</tr>
<tr>
<td>url-open-doc</td>
<td>110</td>
<td>48</td>
<td>25,778</td>
</tr>
<tr>
<td>shutdown</td>
<td>125</td>
<td>67</td>
<td>5,925</td>
</tr>
<tr>
<td>startup</td>
<td>286</td>
<td>237</td>
<td>63,384</td>
</tr>
<tr>
<td>unmarked traces</td>
<td>137</td>
<td>67</td>
<td>38,965</td>
</tr>
</tbody>
</table>

Figure 5.16: Jext: SMQL code

// Events from JEXT Editor
EventSet JextEvents
{
  caption = "Event (JEXT) " ;
  type = {"method-entry",
          "method-exit",
          "endpoint",
          "thread-start",
          "thread-end",
          "module-load",
          "process-start",
          "object-create",
          "process-end") ;
  include (project) = {"jext");
  include (trace) = {"*"); // all traces
  exclude (trace) = {"startup", "shutdown");
  exclude (name) = {"finalize");
  exclude (container) = {"java.",
                         "javax.", "gnu.");
}

// Get java.net.* events (Network events)
EventSet NetEvents
{
  caption = "Net Event (JEXT) " ;
  type = {"method-entry",
          "method-exit",
          "endpoint",
          "thread-start",
          "thread-end",
          "module-load",
          "object-create"
          "process-end") ;
  include (project) = {"jext");
  include (trace) = {"*");
  exclude (trace) = {"startup", "shutdown");
  exclude (name) = {"finalize");
  include (container) = {"java.net.");
}

// Compute the union of JextEvents & NetEvents
AllEvents = JextEvents + NetEvents ;

// Apply the event-analyzer
results = AnalyzeEvents (AllEvents) ;
for Jext. In Figure 5.17, edges represent interaction relationships between features. The dominant features are file-open-doc and bookmark-open-doc, where a new document is created and initialized from the source files. It is expected that other features such as editing, searching, and e-mailing will reference the open documents of a text editor. We also observe that the search-replace feature uses objects that were created by the search feature. This is expected for a search-replace feature, since the search function is needed before the replace function can be applied. The view provides the developer with useful information to reason about the impact of changes without the need to read the source code. For example modifying the ‘replace’ capability of search/replace feature will not have an impact on the search feature, while modifying the search capability will have a direct impact on the search-replace feature. Similarly, modifying the file-open-doc feature will have a direct impact on all of the other features.

![Diagram of feature interaction in Jext](image)

**Figure 5.17: Jext: Feature-interaction view**

Figure 5.18 shows a partial mapping of features to classes. The edges point to the classes that were exercised during the execution of a given feature.

The similarity between features of the Jext editor is shown in Table 5.9, which shows the similarity matrix that is computed by matching the call-graphs for each pair of features.
Figure 5.18: Jext: Feature-implementation view
Figure 5.19: Jext: Result window
### 5.2.3 Jext: Structural Views

These views are constructed as a hierarchy with different levels of abstraction (Figure 5.20). The lowest level represents the object-interaction views. This view is constructed from the execution traces of the program. It serves as the basis for higher level views such as the class-interaction and feature-interaction views. The second level of abstraction includes the class-interaction view. The highest level of the hierarchy represents a program feature.

Figure 5.21 shows a clustered class-interaction view for the file-open-doc feature. This view can be obtained either by selecting the node of the file-open-doc feature from the view shown in Figure 5.17 (obtained by analyzing all features) or by executing the SMQL code segment shown in Figure 5.23. The labels of clusters represent the dominant class within the cluster. Figure 5.22 shows the class-interactions of the cluster labeled as org.jext.JextFrame. This view is created by selecting the nodes representing the cluster in Figure 5.21.

### 5.2.4 Jext: Protocol View

The protocol view of a class is derived from the sequences of method invocations performed on the object instances of the class. A method sequence represents the order in which the methods were invoked on an object during the execution of a program. The set of method

---

Table 5.9: Jext: Feature similarity matrix

<table>
<thead>
<tr>
<th></th>
<th>file-open-doc</th>
<th>search</th>
<th>bookmark-open</th>
<th>bookmark-add</th>
<th>edit-paste</th>
<th>url-open-doc</th>
<th>search-replace</th>
<th>change-lang</th>
<th>email-doc</th>
<th>edit-cut</th>
<th>edit-copy</th>
<th>startup</th>
<th>shutdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>file-open-doc</td>
<td>100</td>
<td>6</td>
<td>68</td>
<td>31</td>
<td>33</td>
<td>42</td>
<td>30</td>
<td>24</td>
<td>28</td>
<td>20</td>
<td>12</td>
<td>18</td>
<td>40</td>
</tr>
<tr>
<td>search</td>
<td>100</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>14</td>
<td>11</td>
<td>14</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>bookmark-open</td>
<td>100</td>
<td>100</td>
<td>35</td>
<td>36</td>
<td>29</td>
<td>51</td>
<td>24</td>
<td>31</td>
<td>21</td>
<td>11</td>
<td>16</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>bookmark-add</td>
<td>100</td>
<td>24</td>
<td>29</td>
<td>42</td>
<td>59</td>
<td>41</td>
<td>36</td>
<td>21</td>
<td>10</td>
<td>21</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edit-paste</td>
<td>24</td>
<td>43</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>35</td>
<td>26</td>
<td>7</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>url-open-doc</td>
<td>100</td>
<td>20</td>
<td>34</td>
<td>18</td>
<td>8</td>
<td>30</td>
<td>7</td>
<td>17</td>
<td>7</td>
<td>10</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>search-replace</td>
<td>100</td>
<td>100</td>
<td>31</td>
<td>39</td>
<td>28</td>
<td>14</td>
<td>9</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>change-lang</td>
<td>100</td>
<td>100</td>
<td>27</td>
<td>32</td>
<td>20</td>
<td>7</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>email-doc</td>
<td>100</td>
<td>100</td>
<td>30</td>
<td>17</td>
<td>14</td>
<td>4</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edit-cut</td>
<td>100</td>
<td>100</td>
<td>54</td>
<td>4</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edit-copy</td>
<td>100</td>
<td>100</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>startup</td>
<td>100</td>
<td>100</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shutdown</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
sequences is processed to approximate the usage scenarios of a class. The protocol view is represented as a set of regular expressions, where each expression represents a usage scenario.

To demonstrate how a developer can use our approach for estimating the class usage scenarios, consider how the **gnu.regexp.RE** class is used in the **Jext** application. The RE class, which is part of the GNU regular expression package, provides an interface for compiling and matching regular expressions. The analysis of the method sequences of the RE class produced three usage scenarios. The association between symbols and methods for the RE class is as follows:

\[
A < \text{init} > \quad \text{Constructor} \\
B \quad \text{match} \\
C \quad \text{isMatch} \\
D \quad \text{chain}
\]

For the RE class the protocol view is described as:

\[
A(D)^{3}(B)^{*} \\
A(C|B|D)^{3}(C)^{*}
\]
Figure 5.21: Jext: Clustered class-interaction view of the open-document feature
Figure 5.22: Jext: Class-interaction of org.jext.JextFrame cluster
/*
 * Jext — Analysis of File-Open-Doc feature
 */

Traces = {"file-open-doc"} ;

EventSet fileOpenEvents
{
  type = {
    "method-entry",
    "method-exit",
    "endpoint",
    "thread-start",
    "thread-end",
    "module-load",
    "process-start",
    "object-create",
    "process-end"}

  include (project) = {"jext"} ;
  include (trace) = traces ;
  exclude (container) = {"java.*", "javax.*", "gnu.*"} ;
}

Analysis = analyzeEvents (fileOpenEvents) ;

Figure 5.23: Jext: SMQL code for the open-document feature
Below is a detailed description on how the above protocol view is derived. The method sequences of the groups that represent each scenario are:

- **Group-1:**

\[
\{A(B)^{178}, AB, A(B)^2, A(B)^7, A(B)^{18}, A(B)^{31}\}
\]

The combined expression of the above sequences is:

\[
A(B)^{1,2,7,18,31,178} \approx A(B)^+ \]

This group represents a usage scenario in which the RE object is created (<init> method), then one or more match methods are invoked. The match method checks if the input in its entirety is an exact match of a regular expression. The scope of the match method is the gnu.regexp package. The isMatch method is essentially similar to the match method with the exception of its public scope, which makes it visible outside of the gnu.regexp package.

- **Group-2:**

\[
\{AC, A, AB, A(C)^2, A(C)^4, A(C)^7, A(C)^{14}, A(C)^{22}, A(C)^{160}, AD\}
\]

The combined expression is:

\[
A(C|B|D)^{0,1}(C)^{0,2,4,7,14,22,160} \approx A(C|B|D)^7(C)^* \]

This group represents a usage scenario in which the RE object is created (<init> method), then an optional invocation to one of the \{match, isMatch, chain\} methods, followed by zero or more invocations of the isMatch method. Note that the chain method is used to add a token that corresponds to part of a regular expression to the internal RE list data structure.
• Group-3:

\{AD(B)^{178}, AD, ADB, AD(B)^2, \\
AD(B)^7, AD(B)^{31}, AD(B)^{18}\} \\

The combined expression is:

\[AD(B)^{0,1,2,7,18,31,178} \approx AD(B)^*\]

This group represents a usage scenario in which the RE object is created (<init> method) followed by an invocation to the chain method, followed by one or more invocations to the isMatch method.

It is worth noting that group-1 and group-3 could be identified as one scenario of the form \(A(D)^?(B)^*\).

5.3 TechReport: Publication Database

The TechReport System is a simple technical report publication management system. TechReport has four features: add publication, search for publication by title, search for publication by author, and update publication. TechReport is a three tier system. The Font-end is a Java client, the middle tier comprises a single Enterprise JavaBeans (EJB) session Bean. The back-end is a relational database (Microsoft SQL Server). Figure 5.24 illustrates the TechReport architecture. This part of the case study demonstrates the capability of our environment for profiling and analyzing distributed systems.

Enterprise JavaBeans (EJB) is the distributed model that is used to implement the TechReport program. The next sub-section describes an overview of the EJB architecture.

Enterprise JavaBeans (EJB) overview

Enterprise JavaBeans (EJB) is an execution environment and a framework for server-side Java components [48]. The EJB component model comprises EJB components, EJB containers, and application servers. An EJB component executes within an EJB container that provides an application context for one or more components. The container is responsible
Figure 5.24: TechReport architecture

for component registration, providing a remote interface for components, component lifecycle, and the coordination of distributed transactions. An EJB application server hosts one or more EJB containers. The EJB server provides resource management services for its container such as threads, memory, database connections, network connections, security, transactions, and persistence. Figure 5.25 illustrates the EJB architecture.

An EJB bean (component) exposes its interface as a CORBA [53] interface (OMG IDL) or as a Java/RMI interface, and Java and non-Java clients can access it remotely. For Java-only it can be accessed through RMI/JRMP, and for non-Java clients it can be accessed either through RMI/IIOP or COM/CORBA/IIOP.

A client does not directly interact with an EJB Bean (component). Instead it interacts with two wrapper interfaces provided by the EJB container that intercepts all requests. The first wrapper interface is the EJB/Home interface that clients use to create instances of a bean. The second wrapper interface is the EJB/Object interface, which clients use to invoke operations on the bean.

The EJB model supports both transient (session) and persistent (entity) objects. For each bean instance the container generates an instance of the context object to maintain information about the bean’s management rules.

5.3.1 TechReport: Build and Profiling

As noted earlier, there is no re-compilation needed for Java applications. To profile the server and client components of the TechReport system, the following steps are taken:
Figure 5.25: Enterprise Java Bean Architecture
1. **Run the LogicalTimeServer**, which is a server component responsible for maintaining a global logical time between the participating profilers. The logical time allows the correct ordering of runtime events when analyzing distributed systems such as TechReport. The additional function of the time-server is to act as a proxy for passing the label of the marked-traces between profilers if the marked-trace is marked as distributed using the Mark-Trace wizard. The script to run the time server is `timeserver.bat`, which contains the following lines:

```bash
CLASSPATH=%SCE_HOME%\jlib\sd.jar
java serg.tools.timeserver.LogicalTimeServer
```

2. **Run the EJB server.** The script to run and profile the EJB/J2EE server is `prof_j2ee.bat`, which is a modification of the original `j2ee.bat` script that comes with the J2EE distribution. The changes to the original script is to add the debugging (`-Xdebug -Xrunjvprof`) options. It is worth noting that the Publication JavaBean of the TechReport should be deployed before running the client. The deployment is performed by running `deploytool` (part of J2EE) and then loading and deploying `ejb_publication.jar`, which contains the JavaBean and the deployment descriptor.

3. **Run the TechReport client.** Profiling the client is performed, simply, by running the following script:

```bash
SET CLASSPATH=pubclient.jar;ejb_publicationsClient.jar
SET JVM_PROPERTIES=-Dorg.omg.CORBA.ORBInitialHost=<Hostname>
jvprof pub.Main
```

Figure 5.26 illustrates a refinement of the software comprehension environment architecture as it pertains to the analysis of the TechReport.
5.3.2 TechReport: Features

Table 5.10 shows a summary of the statistics per feature, which includes the number of classes loaded, the number of methods executed, and the number of events created for each feature on both the client and the EJB-server side. During profiling all features are set to be distributed, so that the trace-name is propagated from the client to the server via the time-server. This allows the tracking of a given feature from the client to the server.

The SMQL code used to derive the software views from the runtime events is listed in Figure 5.27. First, we define two sets of events \texttt{ejbEvents} and \texttt{finalizeEvents}. The \texttt{ejbEvents} set includes events that were created during the execution of the marked-traces from the TechReport client and EJB server. The \texttt{finalizeEvents} set defines the events from the \texttt{finalize()} method. The \texttt{finalize()} method is excluded, by removing \texttt{finalizeEvents} from \texttt{ejbEvents}, to prevent the mis-identification of interactions, since this method is invoked by the garbage collector and the JVM does not guarantee which thread will invoke this method. Finally, we invoke the event analyzer (\texttt{AnalyzeEvents}) to create the dynamic views from the input event set. The returned value, \texttt{analysis}, is a set of sets, where each element (set) represents a \texttt{RelationSet} that defines a view or a part.
Table 5.10: TechReport: Features coverage statistics

(a) Client-side

<table>
<thead>
<tr>
<th>Features</th>
<th>Methods</th>
<th>Classes</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-publication</td>
<td>44</td>
<td>16</td>
<td>1,538</td>
</tr>
<tr>
<td>search-by-author</td>
<td>35</td>
<td>13</td>
<td>910</td>
</tr>
<tr>
<td>search-by-title</td>
<td>31</td>
<td>11</td>
<td>783</td>
</tr>
<tr>
<td>update-publication</td>
<td>63</td>
<td>21</td>
<td>2,353</td>
</tr>
<tr>
<td>unmarked traces</td>
<td>86</td>
<td>24</td>
<td>2,384</td>
</tr>
</tbody>
</table>

(b) EJB server-side

<table>
<thead>
<tr>
<th>Features</th>
<th>Methods</th>
<th>Classes</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-publication</td>
<td>48</td>
<td>11</td>
<td>1,972</td>
</tr>
<tr>
<td>search-by-author</td>
<td>46</td>
<td>12</td>
<td>1,864</td>
</tr>
<tr>
<td>search-by-title</td>
<td>36</td>
<td>12</td>
<td>858</td>
</tr>
<tr>
<td>update-publication</td>
<td>55</td>
<td>13</td>
<td>3,274</td>
</tr>
<tr>
<td>unmarked traces</td>
<td>68</td>
<td>13</td>
<td>3,044</td>
</tr>
</tbody>
</table>

of a view. It is worth noting that the SMQL code here is similar to that for the Jext case study.

Figures 5.28 and 5.29 shows the feature-interaction and feature-implementation views for TechReport. In Figure 5.28, edges represent interaction relationships between features. Figure 5.29 shows the mapping of features to classes. The edges point to the classes that were exercised during the execution of a given feature. The classes are from both the server and the client. For the add-publication feature, the pub.PubClient and pub.CEditPubPanel classes are from the client code, and pub.PubDB and pub.BPbulication classes are from the server code.

Next, we describe the similarity between the features of the TechReport. Table 5.11 and 5.12 show the similarity matrices for the features at the server-side and client-side, respectively. From the tables, it is clear that the add-publication and update-publication features have a high degree of similarity in both the client and the server. The search-by-author and search-by-title features have a high degree of similarity on the client-side.
// Events from TechReport (an EJB application)

projects = {"techreport-client", "techreport-server"} ;

EventSet ejbEvents
{
    caption = "Events for TechReport" ;
    type = {
        "method-entry",
        "method-exit",
        "endpoint",
        "thread-start",
        "thread-end",
        "process-start",
        "object-create",
        "process-end"};
    include (project) = projects ;
}

EventSet finalizeEvents
{
    type = {
        "method-entry",
        "method-exit"
    };
    include (project) = projects ;
    include (name) = {"finalize"} ;
}

// Remove finalize() events
ejbEvents2 = ejbEvents - finalizeEvents ;

// Apply the event-analyzer
Analysis = analyzeEvents(ejbEvents) ;

Figure 5.27: TechReport: SMQL code

Figure 5.28: TechReport: Feature-interaction view
Figure 5.29: TechReport: Feature-implementation view
Table 5.11: TechReport: Feature similarity matrix (Server-side)

(a) Feature similarity with all classes

<table>
<thead>
<tr>
<th></th>
<th>add-publication</th>
<th>search-by-author</th>
<th>search-by-title</th>
<th>update-publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-publication</td>
<td>100</td>
<td>56</td>
<td>52</td>
<td>74</td>
</tr>
<tr>
<td>search-by-author</td>
<td>100</td>
<td>64</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>search-by-title</td>
<td></td>
<td>100</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>update-publication</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Feature similarity without common classes

<table>
<thead>
<tr>
<th></th>
<th>add-publication</th>
<th>search-by-author</th>
<th>search-by-title</th>
<th>update-publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-publication</td>
<td>100</td>
<td>18</td>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td>search-by-author</td>
<td>100</td>
<td>18</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>search-by-title</td>
<td></td>
<td>100</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>update-publication</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.12: TechReport: Feature similarity matrix (Client-side)

(a) Feature similarity with all classes

<table>
<thead>
<tr>
<th></th>
<th>add-publication</th>
<th>search-by-author</th>
<th>search-by-title</th>
<th>update-publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-publication</td>
<td>100</td>
<td>31</td>
<td>36</td>
<td>50</td>
</tr>
<tr>
<td>search-by-author</td>
<td>100</td>
<td>73</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>search-by-title</td>
<td></td>
<td>100</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>update-publication</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Feature similarity without common classes

<table>
<thead>
<tr>
<th></th>
<th>add-publication</th>
<th>search-by-author</th>
<th>search-by-title</th>
<th>update-publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-publication</td>
<td>100</td>
<td>0</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>search-by-author</td>
<td>100</td>
<td>47</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>search-by-title</td>
<td></td>
<td>100</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>update-publication</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.3.3 TechReport: Structural and Remote-interaction Views

These views are constructed as a hierarchy with different levels of abstraction (Figure 5.30). The lowest level represents the object-interaction views. These views are constructed from the execution traces of the program. It serves as the basis for higher level views such as the class-interaction and feature-interaction views. The second level of abstractions includes class-interaction and thread-interaction. The highest level of the hierarchy includes program features.

![Hierarchical view navigation](image)

**Figure 5.30: TechReport: Hierarchical view navigation**

Figures 5.31 and 5.33 show the class-interaction views for the *add-publication* and *search-by-title* features, which involves the TechReport client and EJB-server, respectively. The remote interactions are highlighted using the *endpoint* entity, which is represented as a diamond in the figures. The endpoint label shows the application name, remote host, remote port, local host, and the local port of the endpoint node.

Figure 5.32, shows the class-interaction of the client-side of the *add-publication* feature, this view is part of the view in Figure 5.31.

Figure 5.34 shows the thread-interaction of the TechReport across the client and the server. Nodes represent threads, and an edge between two nodes indicates that the two threads execute concurrently and share some runtime objects. This view helps developers identify concurrent threads that operate on the same objects and highlight areas where
potential race or deadlock conditions may occur in the program. The remote interactions between threads is highlighted by the endpoint nodes (see Figure 5.32) connecting the threads. The details of each thread can be viewed by double-clicking on the thread node, which constructs the class-interaction and call-graph views of the thread. The shared objects between concurrent threads are revealed by double-clicking on the edge connecting two threads.

5.4 Summary

This chapter describes the analysis of three software systems in a case study that demonstrates the effectiveness of our tools. The software systems analyzed in the case study demonstrate various aspects of our research. The analysis of Mozilla demonstrates the effectiveness of the tools to profile and analyze large-scale systems on the Win32 platform that are written in C/C++. Feature-implementation, feature-similarity, module-interaction and class-interaction views based on call-graphs are created during the analysis of Mozilla. The analysis of Jext demonstrates the profiling and analysis of medium-sized Java application. Feature-interaction, feature-implementation and class-interaction views based on object-interactions are created for Jext. The analysis of TechReport demonstrates the profiling and analysis of distributed systems. Feature-interaction, feature-implementation, class-interaction, remote-interaction, and thread-interaction views that are based on object-interactions are created during the analysis of TechReport.
Figure 5.32: TechReport: Class-interaction view of the add-publication feature (Client-side)
Figure 5.33: TechReport: Class-interaction view of the search-by-title feature
Figure 5.34: TechReport: Thread-interaction view
Chapter 6: Conclusions

The major contribution of the research described in this dissertation is the development and implementation of a software comprehension framework that supports the creation of dynamic views to address key challenges in comprehending modern software systems.

The architecture software comprehension framework consists of three subsystems: data gathering, data repository, and modeling/visualization. The framework is implemented as an environment that integrates a set of tools that support the static and dynamic analysis of software written in multiple languages. The environment also supports the analysis of distributed systems, and system that use binary components. Four key challenges in comprehending software systems have been addressed in this research. First, the design and implementation of a data repository for storing static and dynamic data from multi-language applications. Second, the construction of a distributed profiler that collects runtime data from components that execute across a network. Third, the development of a software modeling query language (SMQL) for data retrieval and the implementation of analyzers for creating software views. Fourth, the decoupling of the environment’s subsystems to enhance the extensibility and ease the integration of new tools.

The dynamic software views can model features, remote interactions, thread interactions, software structural, and class protocols. Feature-based views can model feature interactions, implementation, and similarity. The feature-interaction view highlights the dependencies between features. The interaction between features are recovered using profilers that capture the runtime objects referenced during the execution of the features. The feature-interaction view is useful to assess the impact of changing one feature on other features. The view can also help to determine the dominant feature(s) of an application. The feature-implementation view identifies the portions of the source code that implement a
particular feature, as well as highlights the portions of source code that are shared between features. This view can help to assess the impact of source code changes on features. The feature-similarity view shows the similarity between pairs of program features. This view helps identify similar features and, thus, guides engineers to learn about the implementation of a feature by studying similar features. The similarity measure also helps to assess the impact of changing a feature on the rest of the features of the software system. Our two key innovations with respect to feature-based views are: (a) the feature-interaction view, and (b) the method we developed to mark execution traces in order to identify features.

The remote-interaction view helps identify remote invocations between component of a distributed system. This view is unique to our research in identifying remote interactions through dynamic analysis. The thread-interaction view pinpoints to shared classes between threads that may cause deadlock or race conditions for multi-threaded applications. Further work is needed in this area to allow the automatic identification of potential deadlock or race conditions between threads.

The class protocol view describes the order in which the methods are invoked, as well as the set of scenarios that the class is used. Our contribution is attempting to find a compact approximation of the set of method sequences of a class and presenting them as regular expressions. Further work on this view includes the study of groups of collaborated classes that can be considered as a subsystem, and approximate their protocol as a subsystem usage protocol. This is a harder problem, but it will provide more useful information to an engineer, because in general to accomplish a task more than one class is used.

A case study, which consists of three software systems, was used to demonstrate the effectiveness of the software comprehension environment. The software systems analyzed in the case study demonstrate various aspects of our research. The analysis of the Mozilla web browser demonstrates the effectiveness of the environment to profile and analyze large software systems on the Win32 platform that are written in C/C++. The analysis of the Jext text editor demonstrates the environment’s effectiveness in profiling and analyzing medium-sized Java applications. Finally, the analysis of TechReport system demonstrates the environment’s support of profiling and analyzing of distributed systems.
Several opportunities for future work have been identified, namely in the areas of data collection, analysis, and integration with existing IDEs. It is worth noting that most research, including our own, focuses on languages such as C/C++ and Java, and ignores scripting languages (e.g., Visual Basic, PowerBuilder, JavaScript, and JSP). Such languages are often used in the development of information systems. Data collection tools, static and dynamic, for scripting languages and for popular frameworks such as COM+ and Microsoft.Net are needed to provide a comprehensive tool suite to support the analysis of many modern information systems. The creation of software views can be simplified by reducing the reliance on Java to create SMQL filters. The enhancements for SMQL will include conditional statements and pattern-matching rules to automatically derive architectural styles such as pipe-filter, client-server, and other styles. The integration with commonly used IDEs (e.g., Eclipse [19]) will make our tools easily accessible to software developers. Also, to improve the interoperability of our tools, we plan to develop tools that can convert software views into formats acceptable by other case-tools.

With this work we hope to inspire integrated development environment (IDE) developers to add features that employ multi-threaded and distributed profilers in order to create views, such as the ones described in this dissertation, that aid in the comprehension of modern software systems.
Appendix A: Software Modeling Query Language (SMQL)

This appendix describes the syntax of the SMQL language.

\[
Start ::= Declarations || Statements \\
Declarations ::= ImportDecl || EntityDecl || RelationDecl || EventDecl \\
Statements ::= AssignExpr || ProcedureDecl \\
ImportDecl ::= Import JavaClass as Identifier; \\
EntityDecl ::= EntitySet Identifier \\
\{ \\
\quad \{caption = String;\} \\
\quad type = StringSet; \\
\quad (IncludeFilter) * \\
\} \\
EventDecl ::= EventSet Identifier \\
\{ \\
\quad \{caption = String;\} \\
\quad type = StringSet; \\
\quad (IncludeFilter) * \\
\]
RelationDecl ::= RelationSet Identifier

{
  [caption = String;]
  (IncludeFilter)*
  (EntityDecl)*
  (RelationDefs) +
}

IncludeFilter ::= (Include|Exclude)(attribName)

=Expr;

RelationDefs ::= (Include|Exclude)(relationType) =

  { Identifier->Identifier
    (, Identifier->Identifier)* } ;

AssignExpr ::= unaryExpr = Expr ;

Expr ::= AdditiveExpr

AdditiveExpr ::= MultiExpr((+|-)MultiExpr)*

MultiExpr ::= UnaryExpr((&|*)UnaryExpr)*

UnaryExpr ::= ( Expr ) ||

  SetExpr||

  String||

  FunctionExpr||

  ArrayExpr||

  Identifier

FunctionExpr ::= Identifier( ParameterList )

ProcedureDecl ::= FunctionExpr ;

ParameterList ::= UnaryExpr(,UnaryExpr)*
SetExpr ::= \{(Expr, Expr) * \}

ArrayExpr ::= Identifier[String||Integer]

Identifier ::= Letter(Letter||Digit) *

attribName ::= Identifier||String

relationType ::= Identifier||String

JavaClass ::= java: Identifier(.Identifier) *

Letter ::= [A – Za – Z]_

Digit ::= [0 – 9]

String ::= C-Style string (i.e. String)

Integer ::= ([0 – 9])+

Comments are C-Style comments (// and */ */)
Appendix B: User Manual

This appendix describes the software comprehension environment and the set of tools that are part of the environment. It also describes instructions and how to install and use each tool, as well as how to create new analyzers and software views.

B.1 Setup and Installation

Requirements

- The Java development and runtime environment: JDK or JRE 1.4.x.
- The graph drawing tool: dot/dotty [26].
- The software clustering tool: Bunch [42].
- The graph visualization package: JGraph [3].

Download

Download the distribution package from the Software Engineering Research Group at Drexel University web-site (http://serg.cs.drexel.edu) to get a copy of the environment.

Installation

- Create an installation directory. For example:
  - Windows: mkdir c:\mySCE
  - Unix: mkdir $HOME/mySCE

Unzip the distribution file in the installation directory. The directory structure inside the installation directory is shown in Figure B.1:
sce/bin  Executable and scripts.
sce/jlibs  Java jar files.
sce/config  Global configuration files.
sce/schema  SQL scripts to build and initialize the data repository.
sce/data  Collected runtime events data files (xml).
sce/log  Error and trace messages.
sce/ext  Event handler DLLs and extensions.
sce/examples  Examples of programs.
sce/db  The default location for the database files of the repository, if either Derby (formerly called Cloudscape) or HypersonicSQL are used.

- Set the environment variable SCE_HOME to INSTALLATION-DIRECTORY/sce.

- Add sc.jar, sd.jar and bunch.jar to the CLASSPATH. Also, add to the CLASSPATH the packages for the JDBC driver that will be used for the repository if it is other than Derby or HypersonicSQL.

- Create and initialize the repository, by running SceTool (sce/bin/scetool), then build and initialize the repository by selecting Initialize Empty Repository menu item from the File menu. The repository can also be initialized by manually executing the SQL scripts located in the sce/schema directory.

- (For distributed profiling), add and set the TIMESERVER environment variable to the
name of the host that is running the logical time server.

- *(Optional)* Add `sce/bin` to the PATH environment variable.

## B.2 Data Collection Tools

Currently, the environment supports the following data extractors:

- **Wdbg**: A Win32 C/C++/VB profiler that is implemented using the Win32 debugging API.

- **WinProf library**: This library contains the function entry and exit handlers and two additional dynamically-linked libraries to allow the instrumentation and profiling of Win32 program. The difference between using the library and using the **Wdbg** profiler is that a re-compilation of the program is required for the library to instrument the source code. Also, this method of profiling is more suitable for large programs, since it provides better (and relatively acceptable) performance over the **Wdbg** profiler.

- **SymDbg**: A Win32 scanner that extracts source code entities from the compiled binaries (DLL and EXE) of a program.

- **JvProf**: A Java profiler that is implemented using the JVMDI and JVMPI interfaces.

- **Jsa**: A Java static analyzer that is implemented using the BCEL [25] package.

### B.2.1 Wdbg: Profiler for Windows Applications

**Wdbg** is a C/C++ local profiler that is implemented using the Windows debugging API. The profiler acts as a debugger process for Win32 binary executables. First, it reads the debugging symbol table and all of the imported dynamically-linked libraries (DLL), then it sets breakpoints on all identified functions. Inclusion and exclusion filters on source code files, DLL modules, and functions are supported to limit the amount of runtime data that is gathered.

To facilitate distributed profiling and the identification of remote-interactions, a network interceptor was implemented as part of the **Wdbg** profiler. The network interceptor
intercepts calls to and from the winsock dynamic linked library on the Win32 platform. The winsock library is the Windows implementation of the TCP/IP protocol API (known as the Socket Interface). By default, if the winsock library is used, the profiler sets breakpoints on the connect, accept, read, and write functions. The socket value, remote host, and remote port number are read from the stack frame of the watched functions, while the local port is read using the wrapper functions of the intercepted functions. The interceptor is implemented as a profiler extension in a separate dynamic link library (DLL), which is loaded into the profiled program’s process memory.

Figure B.2 illustrates an overview of the static and dynamic view of the Wdbg components. The Wdbg profiler consists of three components:

1. **The profiler executable module** (Wdbg.exe), which loads the profiled-program, loads the profiler extensions (such as the network interceptor) into the profiled program’s process space, and loads the data collection adapter.

2. **The profiler extensions**, which are separate dynamically-linked libraries. These extensions are loaded into the profiled-program process space, since they need access to functions and parameters that are associated with the program process. As mentioned earlier, the network interceptor is implemented as an extension. When the profiler loads the program, it allocates a block of memory in the process spaces, and writes assembly instructions to load the extensions. Upon receiving the program entry breakpoint from the process, the profiler sets the instruction pointer to the first byte of the added instructions to force the process to execute, which results in the loading of these extensions. For example, when the network interceptor library is loaded, it iterates on all imported functions in the program and searches for the winsock function, and forward them to the wrapper functions. The wrapper functions read the socket parameters (remote-host, remote-port, local-host, and local-port) and sends them to the profiler before invoking the original winsock function. Another example of an extension is the mark-trace wizard, which sends the start and end of the trace name to the profiler.
3. **The data collection adapter**, which is loaded and initialized by the profiler. The profiler sends all runtime event to the data collection adapter. The current adapter stores runtime events in the XML format. These events are then loaded into the repository using the `dbimport` tool.

Below is an outline of the key features of the *Wdbg* profiler, which the user can define in the profiler configuration files:

- Include and exclude filters at the function, source file or module level.
- Runtime events include process start/end, thread start/end, method entry/exit, module load/unload, and endpoint events.
- Custom event-handlers can be implemented as separate DLLs and loaded at runtime.
- Custom extensions are DLLs that are attached to the profiled program’s process memory. The network interceptor is implemented as an extension. Extensions are
useful to implement library detours or wrappers without the need to replace the existing DLLs.

The profiler uses two configuration files: a global file, which is located in the sce/config/ directory of the installation directory, and a local file. The global file is applied to all profiled processes, while the local file is applied to a particular process if it is specified. Both files has the same format. An example of the Wdbg configuration file is shown in Figure B.3.

**Wdbg: Usage**

```
Wdbg [options] program
```

Where `options` can be zero or more of the following command line options:

- `--conf`  Global configuration file, which is an XML file that describes common configurations that are applied to all profiled applications. If specified, it overwrites the file in sce/config/ directory.
- `--lconf` Local configuration file that is applied to the current profiled program.
- `--log` Log path for error and trace messages
- `--o` Output file of the events. If not specified, the file name is `<ProgramName>−<Process-ID>−<Format>.xml`
- `--f` Format (XML or tab). Default format is XML.

**Wdbg: Setup**

1. Unzip the archive into the installation directory (i.e., WINPROF)

2. Set the environment variable `SCE_HOME` to WINPROF/sce

3. Place the WINPROF/sce/bin path into the PATH environment variable.

The directory structure of the installation directory is:

```
sce/bin       Executable (Wdbg)
sce/config    Global configuration files
sce/data      Collected runtime events data files (xml)
sce/log       Error and trace messages
sce/ext       Event handler DLLs and extensions
```
Figure B.3: Wdbg: An example of the profiler configuration file
B.2.2 WinProf: Win32 Profiler Library

This library contains the function entry and exit handlers and two additional dynamically-linked libraries for the instrumenting and profiling Win32 programs. The difference between using the library and using Wdbg profiler is that a re-compilation of the program is required in order to use the DLL. Also, this method of profiling is more suitable for large programs, since it provides better (acceptable) performance over the Wdbg profiler.

Using the profiler library, profiling is performed using code instrumentation by inserting function handlers to intercept function-entry and exit events. The implementation of function-entry handler uses the Microsoft C++ compiler option (/Gh), which enables a function call hook, while the function-exit handler was implemented by manipulating the call stack and forcing the program to invoke our own function-exit handler. The function entry and exit events are then passed to the data collection module.

This approach is an alternative solution to the Wdbg profiler if the source code is available and the application’s performance is degraded significantly using Wdbg.

WinProf: Library usage

The Win32 profiler library can be used with programs that were compiled using the Microsoft C++ compiler. The following are the basic instructions to use the library:

- Compile the source code using the /Gh compiler option.

- Add the penter.lib static library to the project libraries. The library is located in the sce/ext directory.

- Build the project (by running the linker) to produce the project binaries.

Now the project is instrumented and ready for profiling. To run the program, and start collecting runtime data, set the following environment variables:

```
set GH_PROFILE_ENABLED=1
set GH_HANDLER_DLL=D:\Distribution\sce\ext\PEnterHandler.dll
Set TRACE_MARKER_LIBRARY=D:\Distribution\sce\ext\TraceMarker.dll
set GH_OUTPUT_PATH=D:\Distribution\sce\data
```
The `GH_PROFILE_ENABLED` variable enables \((value = 1)\) or disables \((value \neq 1)\) profiling, when running Mozilla. The `GH_HANDLER_DLL` and `TRACE_MARKER_LIBRARY` variables specify the full path of the data collection libraries. The `GH_OUTPUT_PATH` variable specifies the location where the collected data is stored, the data is then imported into the repository using the `dbimport` tool. (Note that D:\Distribution\sce\ is the tools installation directory).

### B.2.3 JvProf: Java Profiler

`JvProf` is a Java-based profiler that is implemented using the Java Virtual Machine Profiler Interface (JVMPI) [71] and the Java Virtual Machine Debugger Interface (JVMDI) [70]. The two interfaces complement each other. For example, capturing the runtime object references of class instances is supported in the JVMPI interface and not supported by the JVMDI interface, while watching the field access and modification events is supported by JVMDI but not supported by JVMPI.

`JvProf` monitors the following events:

- **JVM Initialize and death events**: These events are triggered when the JVM is initialized or terminated.

- **Class Load and Unload events**: these events are triggered when the JVM loads or unloads a Java class.

- **Method Entry and Exit events**: these events are triggered upon the entry or exit of a method.

- **Thread Start and End events**: these events are triggered upon the starting or stopping threads.

- **Field Access events**: these events are triggered when fields are accessed (read). At this point, only selected classes of `java.net` package are monitored for field access.

- **Exception events**: these events are triggered when an exception is caught or raised.

- **Object create, delete and move events**: these events are triggered when an object is created, deleted, or moved \((i.e.\ re-allocated due to swapping to and from disk)\).
In addition to the above events, endpoint events are generated whenever a potential remote call is detected. The profiler watches for method call events from instances of the java.net.SocketInputStream and SocketOutputStream classes. The endpoint parameters are determined by reading the field values of the SocketImpl class. The instance of SocketImpl, which is declared in the SocketInputStream and SocketOutputStream classes, holds the values of the remote and local network addresses and the port numbers. Note that Java RMI and EJB use these classes to implement remote invocations.

**JvProf: Usage**

```
java -Xdebug -Xnoagent -Xrunjvprof:[options] [main-class | [-jar jar-file]]
```

OR

```
jvprof [main-class|-jar jar-file]
```

The JvProf options are name-value-pairs separated by a colon:

- `fm=true` Instructs JvProf to watch for field modification events. (Default value is false)
- `fa=true` Instructs JvProf to watch for field access events. (Default value is false)
- `rmiServer=hostname` Instructs JvProf to use the RMI registry on hostname. (Default value is localhost)
- `rmiServerPort=port-number` Instructs JvProf to use the RMI registry on port-number. (Default value is 1099)
- `adapter=java-class` Instructs JvProf to use the java-class as the data collection adapter. Default value is `serg.profiler.javadapter.XMLAdapter`.
- `TimeServer=hostname` Instructs JvProf to use the Time Server running on hostname. (Default value is localhost or the host defined in TIMESERVER environment variable)
- `config=fullpath-of-config-file` Instructs JvProf to use the specified configuration file. If the environment variable `SCE_HOME` is set, JvProf will use `SCE_HOME/conf/JvProf.conf` file.
- `debug=true` Instructs JvProf to operate in verbose mode. (Default value is false)

JvProf uses a configuration file that defines include and exclude filters for classes. In addition, it defines the socket classes, which are used by the network interceptor. A sample of the configuration settings are shown below:

```
# Note: names are case-sensitive
```
JvProf: Setup

To set JvProf on Linux, perform the following steps:

- Add $SCE_HOME/lib to your LD_LIBRARY_PATH environment variable:
  If you use a Bourne shell-derivative (sh/ ksh/bash), add the following lines to the end of your .profile:

    CLASSPATH=$SCE_HOME/jlib/sd.jar:$CLASSPATH
    LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SCE_HOME/lib
    export LD_LIBRARY_PATH CLASSPATH

  If you use a c-shell-derivative (csh/zsh/tcsh), add the following lines to the end of your .cshrc:

    setenv LD_LIBRARY_PATH "$LD_LIBRARY_PATH:$SCE_HOME/lib"

To setup JvProf on Windows, copy the SCE_HOME\lib\jvprof.dll library to the WINNT\System32 directory. On both Windows or Linux, add the sd.jar file to the CLASSPATH.
B.2.4  Jsa: Java Static Analyzer

Jsa is a static analyzer for Java programs implemented using BCEL package [25]. It collects the following information from a Java class or a Java jar file:

• Entities:
  – Package information.
  – Class information: Methods, fields, implemented interfaces, and super class.
  – Method information: name and signature.
  – Field information: name and signature.

• Relations
  – The invokes relationship between two methods.
  – The reads and writes relationship between a method and a field.
  – The defines relationship between a class and a method. The define relation indicates that the method is implemented by the class.
  – The declares relationship between a class and a method or field. And between an interface and a method. The declare relation indicates that the class/interface does not provide an implementation of the method.

Jsa: Usage

\[ \text{jsa } [-f \text{ JavaFile}] [-p \text{ ProjectName}] \text{ serg.sc.jsa.Main} \]

If JavaFile or ProjectName are not provided in the command line, the user will be prompted for them using the dialog box shown in Figure B.4. The JavaFile is the fullpath for a Java class file (.class) or jar file.

B.2.5  SymDbg: Win32 Static Analyzer

The SymDbg is a static analyzer that extracts symbolic information about Win32 binaries (executables and dynamically linked library files). This analyzer only collects entities (such
as classes, modules and functions), and does not collect static relationships. The purpose of this analyzer is to collect the entities in the program, so that the collected information can be used in conjunction with the data collected from the profiler to assess program coverage.

**SymDbg: Usage**

`symdbg.exe -o output-file [[-m module-filename] | [-p modules-path]]`

The user can either specify a single module (.exe or .dll file) or a path where the binary files resides to scan all files in the specified directory.

**B.3 Distributed Profiler**

Modeling the remote runtime interactions between more than one application requires profilers that can intercept network calls (e.g., socket or SocketStream calls) to determine the communication endpoints between the participating distributed components. Profiling a distributed system is different from profiling a traditional system. First, many profiling tools must be in place, because of the physical distribution of the components. Second, a logical clock server must exist to synchronize component interactions, because of the lack of a central clock. Third, a central facility is needed to collect data from the various profilers at different locations. Figure 3.2 illustrates the data gathering subsystem of the SCE Tool.
We implemented network-call interceptors in the **Wdbg** (for Win32-based applications) and **jvprof** (for Java applications) local profilers. Running either or both profilers in conjunction with the logical time server forms a distributed profiler for collecting runtime data from distributed applications written in Java and Win32 C/C++/VB.

Figure B.5 illustrates a typical configuration of the profilers to construct a distributed profiling environment using **Wdbg** and **jvprof** profilers. In addition to identifying remote-interactions, a user can mark traces of execution using the mark-trace wizard. The traces can be marked as distributed to instruct the time-server to pass the trace-name from one program to another. This capability allows users to profile features that cross the boundary of the process and identifies the code that was executed at the client and the server side of the feature.

**Figure B.5: Distributed profiler**

Profiling a distributed application is similar to profiling a stand-alone application, with the exception of running the logical time server. The steps to profile a distributed application are outlined below:

1. Run the logical time server. The script for the time server is `sce/bin/timeserver.bat`,
which contains the following code:

```java
CLASSPATH=%SCE_HOME%/jlib\sd.jar;%CLASSPATH%
java serg.tools.timeserver.LogicalTimeServer
```

2. Run each distributed component (client or/and server) using the appropriate profiler.

3. Import the data gathered from each component into the repository using the `dbimport` tool.

### B.4 SDTool: Sequence Diagrams

SDTool is a UML sequence diagram visualization tool. The tool can be used in two modes. The first mode is as a stand-alone tool, where it reads its input data from an XML file, which contains the runtime events collected during a program. The second mode is as an SMQL visualizer that can be invoked from SMQL code by attaching the `SceEventSequVisualizer` class to an event set as described in Section 3.4.

**SDTool: Usage**

```java
java serg.sc.sd.Main
```

### B.5 Running SceTool

In this section, we describe how to profile a Java program and use the SceTool to analyze the runtime events. We use the KWIC (Keyword in context) program to illustrate the steps. The KWIC example is included in the distribution package of our tools.

**KWIC: Profiling**

To profile the KWIC example run the following command:

```sh
jvprof -jar kwic.jar
```
When the profiler is loaded it prompts the user for the name and location file for storing the runtime events.

**KWIC: Import data into repository**

Run the `dbimport` tool, located in `sce/bin` directory, to import the runtime events into the repository. Figure B.6 shows the box for `dbimport` tool: type the project name (e.g. for example ‘kwic-rt’), and select the XML file of the runtime events. Note that the connection parameters for the repository may need to be adjusted to match the JDBC driver used to connect to the repository.

![Figure B.6: KWIC: Importing runtime data](image)

**KWIC: Run Scetool**

We are ready to analyze the KWIC program. To start `SceTool`, run the `scetool` script in the `sce/bin` directory. When `SceTool` is loaded, the initial screen looks like the one shown in Figure B.7.

Open an SMQL editor windows (from the main menu: Query – > New SMQL Editor), then load the SMQL code for the KWIC programs. The sample code file is `kwic-rt.smql` and is located in `sce/tests` directory. Click on the execute button to run the code.
Figure B.7: KWIC: Running SceTool
Figure B.7 shows the **SceTool** after loading the SMQL code, and Figure B.9 shows the tool after executing the code.

![Software Comprehension Environment (SCE) interface](image)

```cpp
// Analyzing the KWIC program

// Define the set of runtime events

EventSet Events
{
    type = {
        "method-entry",
        "method-exit",
        "thread-start",
        "thread-end",
        "module-load",
        "object-create",
        "object-free"
    }
    // Include events from "kwic" program
    include [project] = {"kwic.rt"};
}

// Use the analyzeEvents() filter to analyze the events
EventAnalysis = analyzeEvents(Events);
```

**Figure B.8: KWIC: SceTool after loading the SMQL code**

To visualize any of the views, simply, double-click on the tree item representing the view. For example, double-clicking on the item labeled `<interact> kwic - rt : T49086720[54]`, will show the view shown in Figure B.10.
Figure B.9: KWIC: SceTool after executing the SMQL code
Figure B.10: KWIC: Viewing software views
Writing an SMQL filter

As stated earlier, in SMQL, software views are created as filters. An SMQL filter is an implementation of the IFilter interface, or an extension of the AbstractFilter abstract class. Figure B.11 shows the Java source code for the IFilter interface and the AbstractFilter class.

```java
package serg.sc.filter;

import serg.sc.repository.Repository;

public interface IFilter
{
    public void compute() throws Exception;
    public void setRepository(Repository repository) throws Exception;
    public void setInput(Object val, int index) throws Exception;
    public Object getOutput(int index) throws Exception;
    public int getOutputCount();
    public String getDescription();
    public String getName();
}
```

**Figure B.11: Java source code for IFilter interface**

The methods of the IFilter interface are described below:

- `getName` and `getDescription` methods return the name and description of the filter, respectively.

- `setInput` method is invoked by SMQL to set the inputs of the filter. These inputs can be EntityList, RelationList, EventList, a generic set or a string.

- `setRepository` method is invoked by SMQL to set the reference for the repository to allow the filter to access the repository if needed.

- `compute` method is invoked by SMQL to instruct the filter to perform its computation.

- `getOutput` method returns an object reference that holds the output of the filter. The
package serg.sc.filter;

import java.util.*;
import serg.sc.repository.*;

abstract public class AbstractFilter implements IFilter {

  protected String m_Name = null;
  protected String m_Desc = null;
  // Holds the input arguments
  protected ArrayList m_InputsList = new ArrayList();
  protected Repository m_Repository = null;

  public AbstractFilter(String name, String desc) {
    m_Name = name;
    m_Desc = desc;
  }

  public void setRepository(Repository repository) throws Exception {
    this.m_Repository = repository;
  }

  public void setInput(Object val, int index) throws Exception {
    m_InputsList.add(index, val);
  }

  public String getDescription() {
    return m_Desc;
  }

  public String getName() {
    return m_Name;
  }
}

Figure B.12: Java source code for the AbstractFilter abstract class
filter may return more than one object. (The current behavior only allows for one output).

- **getOutputCount** method returns the number of outputs that the filter can provide.
  
The default value is one.

Next, we describe an implementation of an SMQL filter that groups objects into their classes. Figures B.16 and B.17 show the Java source code for the `GroupObjects` class, which extend the `AbstractFilter` abstract class to create a new software view, which groups the object insurances of each class in the program. The `compute` function first invokes the `verify` method to ensure that the input is an event set (instance of `EventList` class), then invokes the `createObjectsGroups` method to group object instances per each class.

To use the `GroupObjects` filter in the SMQL code, a user can import the Java class in the SMQL code as shown in Figure B.13. Executing the SMQL code, shown in Figure B.13, will produce the results shown in Figure B.14. Double-clicking on the `class[12]: objects created per class` item, in the result tree, will produce the tree view shown in Figure B.15.

A user-defined SMQL filter can be made global by adding the filter in the SMQL configuration file (`smql.xml`), which is located in `sce/config` directory, by adding the following line in the `smql.xml` file:

```xml
<AddFilter name="GroupObjects" class="java:serg.sc.examples.GroupObjects" />
```

The `name` attribute defines the name of the filter by which it will be invoked from the SMQL code. The filter name can be different from the class name. The `class` attribute defines the fully qualified Java class name. The class defining the SMQL filter must be added to the `CLASSPATH` to be used successfully. Global SMQL filter does not need to be imported in the SMQL code.

**B.6 Summary**

In this Appendix, we described the software comprehension environment tools including installation and usage instructions. The tools include the data extractors, data import, and the analysis tools. The data extractors include: A Win32 profiler (`Wdbg`), Win32 profiler
// Analyzing the KWIC program

// Import the GroupObjects filter
import javaserg.sc.examples.GroupObjects as GroupObjects;

// Define the set of runtime events
EventSet Events {
    type = {
        "method-entry",
        "method-exit",
        "thread-start",
        "thread-end",
        "module-load",
        "object-create",
        "object-free"
    };
    // Include events from "kwic" program
    include (project) = {"kwic-rt"};
}

// Use the GroupObjects() filter to group instance of classes
classes = GroupObjects(Events);

// Use the analyzeEvents() filter to analyze the events
EventAnalysis = analyzeEvents(Events);

Figure B.13: Importing GroupObjects class in the SMQL code
Figure B.14: SMQL results from executing the GroupObjects filter
Figure B.15: The view resulted from executing the GroupObjects filter
package serg.sc.examples;

import serg.sc.filter.*;
import serg.sc.repository.data.*;
import java.util.*;

/**
 * ObjectsPerClass - Groups objects created by each class in the input event-list.
 */
public class GroupObjects
    extends AbstractFilter
{
    private EntityList m_Classes = null;
    private EventList m_Events = null;

    public GroupObjects()
    {
        super("serg.sc.examples.ObjectsPerClass",
             "Groups objects created by each class");
        m_Classes = new EntityList();
        m_Classes.setName("Objects/Classes");
        m_Classes.setDescription("Objects created per class");
    }

    public int getOutputCount()
    {
        return 1;
    }

    /**
     * @param index int: The index of the output (Default = 0)
     * @throws Exception
     * @return Object: The output of the filter (m_Classes EntityList)
     */
    public Object getOutput(int index) throws java.lang.Exception
    {
        return m_Classes;
    }

    /**
     * Implements the IFilter.compute() method
     * @throws Exception
     */
    public void compute() throws java.lang.Exception
    {
        if (verify() == false)
        {
            throw new Exception(
                "Invalid input argument -- an EventSet is required");
        }
        createObjectsGroups(); // Counts object instance of each class
    }

    Figure B.16: Java source code for the ObjectsPerClass class
private boolean verify()
{
    if (m_InputsList.get(0) instanceof EventList)
    {
        m_Events = (EventList) m_InputsList.get(0) ;
        return true ;
    }
    else
    {
        return false ;
    }
}

private void createObjectsGroups()
{
    HashMap classesMap = new HashMap() ;
    for (int k = 0 ; k < m_Events.size() ; ++k)
    {
        ProfilerEvent event = (ProfilerEvent) m_Events.get(k) ;
        if (event.getTypeName().equals(AttrConstants.EVENT_OBJ_CREATE))
        {
            Entity classEntity
                = (Entity) classesMap.get(event.getEntity().getContainer()) ;
            if (classEntity == null)
            { // Create a class entity
                classEntity = new Entity("class") ;
                classEntity.setName(event.getEntity().getContainer()) ;
                classesMap.put(event.getEntity().getContainer(),
                                classEntity) ;
                // Add class to the classes list.
                m_Classes.add(classEntity) ;
            }
            // Add the object Entity to the class-children list
            classEntity addChild(event.getEntity()) ;
        }
    }
    classesMap.clear() ;
    classesMap = null ;
}
library, Java profiler (JvProf) Java static analyzer (Jsa) and Win32 static binary analyzer (SymDbg). The data import tool (dbimport is used to import program data by the data extractors into the repository. The analysis tool SceTool is the main tool for analysis and visualization. At the core of SceTool is the SMQL parser. Through examples, we described how to use SceTool as well as how to write Java extensions for SMQL.
Bibliography


Vita

Maher M. Salah

Background

Maher Salah has over 15 years of technical experience in software development. Mr. Salah received a B.S degree in Electrical Engineering (with Distinction) from Birzeit University in 1989, M.S. degree in Electrical Engineering from Drexel University in 1993, M.S. degree in Computer Science from Drexel University in 1996, and he is anticipating a Ph.D. degree in Computer Science from Drexel University in 2005. His research interest includes software comprehension, software design and architecture, and component-based and distributed systems. He published several refereed articles, and he holds five US patents.

Publications


Patents
