Dynamic Analysis of Software Systems using Pattern Mining

Dynamic Analysis of Software Systems based on Sequential Pattern Mining

By
HOSSEIN SAFYALLAH, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
M.A.Sc.

McMaster University

DEGREE:	MASTER OF APPLIED SCIENCE (2006)
DEPARTMENT:	Computing and Software
University:	McMaster University, Hamilton, Ontario
TITLE:	Dynamic Analysis of Software Systems based on Sequential Pattern Mining
AUTHOR:	Hossein Safyallah, B.Sc.
SUPERVISOR:	Dr. Kamran Sartipi

NUMBER OF PAGES: xi, 81

Abstract

Software system analysis for identifying software functionality in source code remains as a major problem in the reverse engineering literature. The early approaches for extracting softwares functionality mainly relied on static properties of software system. However the static approaches by nature suffer from the lack of semantic and hence are not appropriate for this task.

This thesis presents a novel technique for dynamic analysis of software systems to identify the implementation of certain software functionalities known as software features. In the proposed approach, a specific feature is shared by a number of task scenarios that are applied on the software system to generate execution traces. The application of a sequential pattern mining technique on the generated execution traces allows us to extract execution patterns that reveal the specific feature functionalities. In a further step, the extracted execution patterns are distributed over a concept lattice to separate feature-specific group of functions from commonly used group of functions. The use of lattice also allows for identifying a family of closely related features in the source code. Moreover, in this work we provide a set of metrics for evaluating the structural merits of the software system such as component cohesion and functional scattering. We have implemented a prototype tool kit and experimented with two case studies Xfig drawing tool and Pine email client with very promising results.

Acknowledgements

I extend my sincere gratitude and appreciation to many people who made this masters thesis possible. I wish to express my sincere thanks to my thesis supervisor, Dr. Kamran Sartipi, for kindly providing guidance throughout the development of my study and preparation of this thesis. I am greatly indebted to Dr. Kamran Sartipi for his immense help in planning and executing this work in time. Gratitude also goes to my second adviser, Dr. Jiming Peng, whose help and suggestions helped me during my study.

My sincere thanks are due to the official referees, Dr. Frantisek (Franya) Franek, and Dr. Mark Lawford, for accepting reviewing this thesis.

Finally, I would like to thank all whose direct and indirect support helped me completing my thesis in time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software system analysis for extracting system functionality remains as a major problem in the reverse engineering literature. The early attempts for extracting software functionality mainly had a static nature and were centered on searching for patterns of the system functionality based on program templates in a knowledge base [17]. However, static analysis suffers from the lack of enough semantics and hence is not appropriate for functionality recovery. The static approaches are mostly useful for extracting the structure of software systems and support specific reverse engineering activities such as re-documentation, restructuring and re-engineering.

There is a growing attention towards the dynamic aspects of software systems as a challenging domain in software reverse engineering [24, 13]. Dynamic analysis deals with task scenarios that formulate the user-system interactions in an informal or semi-formal manner. The approaches to dynamic analysis cover areas such as performance optimization [23], software execution visualization [21], and feature to code assignment [12], where in this work, we address the latter problem. Typically, to understand the implementation of a certain feature of a system, maintainers refer to the documentation of the software system. However, in many cases the mapping of features to the source code is poorly documented and one has to review the entire source code to obtain the required knowledge for this task. In this thesis, we propose a novel approach to dynamic analysis of software systems, in order to identify the implementation of the software features in the source code. In this context, dynamic analysis is performed by executing a group of well-defined task scenarios on the software system and by observing the execution results. Dynamic analysis with its characteristics to extract system functionality has several challenges compared to static analysis: i) in static analysis usually a complete set of software facts are generated through parsing or lexical analysis of the source code based on a domain model, whereas in dynamic analysis only a small subset of the possible dynamic traces are extracted; ii) obtaining meaningful knowledge from the extracted execution traces is a difficult task that restricts the applicability of the dynamic analysis; and iii) the large sizes of the execution traces caused by program loops and recursions may disable the whole dynamic analysis.

In this work, we define and execute a set of task scenarios with a specific shared feature on the software system in order to generate execution traces. The application of a sequential pattern mining algorithm on the extracted execution traces allows us to obtain high-frequency patterns of functions. In a further step, we analyze the frequently appearing patterns, in order to identify the implementation of the software features in the source code. Finally, in a post-processing step we separate the more general patterns(e.g., starting/terminating operations and common utility functions) from feature-specific patterns.

Upon identifying the implementation point of a certain software feature (i.e. the group of feature-specific functions), we assess the impact of the feature on a portion of software structure that contributes to implement this feature. The proposed structural assessment directly represents the cohesion of module(s) implementing a specific feature; this measure of cohesion is much closer to the original definition of cohesion ("relative functional strength of a module" [22]) than using static structural techniques such as inter-/intra-edge connectivity of the components. Furthermore, each group of core functions that implement a feature can be used to incorporate semantics into the existing software architecture recovery techniques [25].

1.1 Problem Description

Software maintenance is the major activity in the software system life cycle and has a critical importance in maintaining both legacy and newly developed software systems. Software maintenance consists of activities including: *corrective* maintenance to diagnosis and correct the errors, *adaptive* maintenance to modify the software system to properly interface with changing environments (hardware and software), *perfective* maintenance to enhance the functionality of the software, and finally *preventive* maintenance to improve the future maintainability and reliability of the software system.

A prerequisite for each of the above mentioned activities is a comprehensive understanding of the whole software system including its design and and run-time aspects. Early attempts for program understanding mostly have been focused on static aspects of the software system based on entities and dependencies in the source code [34]. However static analysis suffers from lack of semantics and is unable to extract the runtime behavior of the software, thus it can not address problems that have a dynamic nature such as identifying the implementation point of the software features, finding the execution bottlenecks and/or the less frequently used part of the system, and understanding the interactions among different software components. Based on the above discussion, we define the problem of this study as:

devising required process, techniques, and supporting tools for identifying the implementation of the functional aspects of a software system in the source code as a means to incorporate semantics into static analysis techniques.

1.2 Proposed Solution

This thesis presents a dynamic analysis approach for identifying the implementation of software features that is based on the frequent patterns of function calls in execution traces of the software system. It also proposes an evaluation metric for assessing the structural merits of the software system based on the degree of functional scattering of the software features among the structural modules.

1.2.1 Proposed Framework

Figure 1.1 illustrates different steps of the proposed framework for assigning software features onto the system modules. The framework provides a means for reducing the large sizes of execution traces, takes advantage of the relation discovery power of *data mining* and *concept lattice analysis*, and allows us to measure the impact of individual features on the structure of the system.

This process consists of four stages: *Execution trace extraction; Execution pattern mining; Execution pattern analysis*; and *Structural evaluation.* In the rest of this section these stages are briefly described.

- *Execution trace extraction*: important features of a software system are identified by investigating the system's user manual, on-line help, similar systems in the corresponding application domain, and also user's familiarity with the system. A set of relevant task scenarios are selected that examine a single soft-

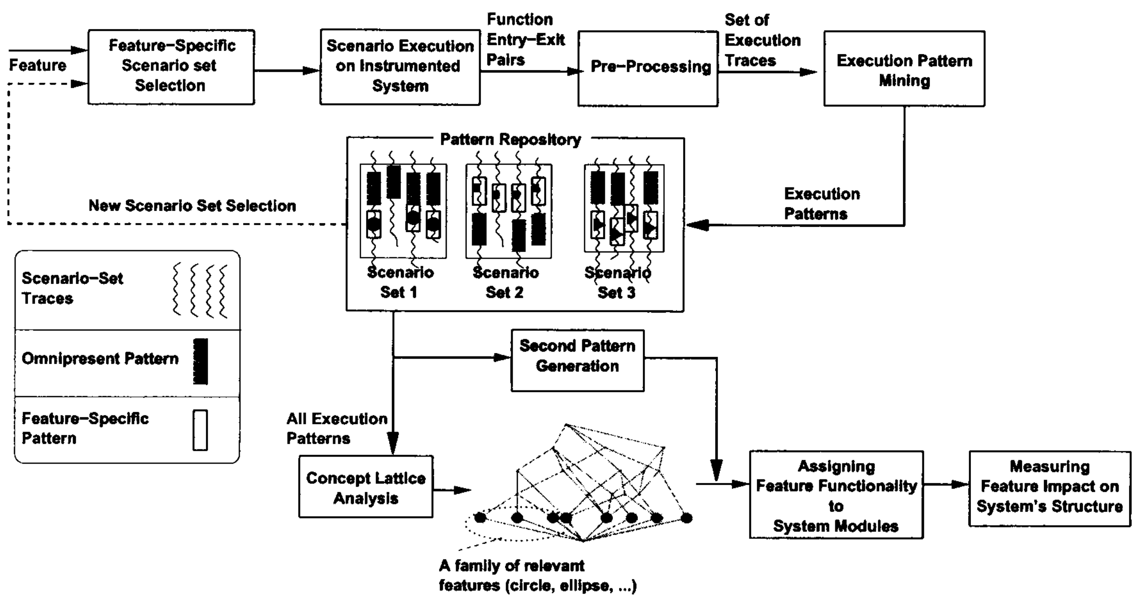

Feature
Feature-Specific Scenario set Selection
Scenario Execution on Instrumented System
Function Entry-Exit Pairs
Pre-Processing
Set of Execution Traces
Execution Pattern Mining
Pattern Repository
New Scenario Set Selection
Execution Patterns
Scenario Set 1
Scenario Set 2
Scenario Set 3
Scenario-Set Traces
Omnipresent Pattern
Feature-Specific Pattern
Second Pattern Generation
All Execution Patterns
Concept Lattice Analysis
Assigning Feature Functionality to System Modules
Measuring Feature Impact on System's Structure
A family of relevant features (circle, ellipse, ...)

Figure 1.1: Proposed framework for identifying the implementation of the functional aspects of a software system in the source code as a means to incorporate semantics into static analysis techniques.

ware feature. We call this set of scenarios as *feature-specific scenario set.* For example, in the case of a drawing tool software system, a group of scenarios that share the "move" operation to relocate a figure on the computer screen would constitute such a feature-specific scenario set. In the next step, the software under study is instrumented[1] to generate function names at the entrance and exit of a function execution. By running each feature-specific scenario against the instrumented software system a sequence of function invocations are generated in the form of *entry/exit pairs.* To make the large size of the generated traces manageable, in a *preprocessing* step we transform the extracted entry/exit pairs into a sequence of function invocations and also remove all redundant function calls caused by the cycles of the program loops. The trimmed execution traces

[1]Instrumentation refers to the process of inserting particular pieces of code into the software system (source code or binary image) to generate a trace of the software execution.

are then fed into the execution pattern mining engine in the next stage. The preprocessing operation will be discussed in more details in Section 5.3.1.

- *Execution pattern mining*: in this stage, we reveal the common sequences of function invocations that exist within the different executions of a program that correspond to a set of task scenarios. We apply a sequential pattern mining algorithm on the execution traces to discover such hidden execution patterns and store them in a *pattern repository* for further analysis. This stage will be discussed in more details in Section 4.1.

- *Execution pattern analysis*: each execution pattern is a candidate group of functions that implement a common feature within a scenario set. We employ a strategy to spotlight on functions in execution patterns corresponding to specific features within a group of scenario sets. This is performed by identifying those patterns that are specific to a single software feature within one scenario set (namely *feature-specific patterns*). Similarly, we identify the patterns that are common among all sets of scenarios (namely *omnipresent patterns*). In Figure 1.1 a sketch of the scenario-set execution traces and *feature-specific / omnipresent* patterns are shown. Even for a specific feature, a large group of execution patterns are generated that must be organized (and some must be filtered out) to identify core functions of a feature. We employ two different mechanisms for this purpose: concept lattice analysis and second sequential pattern mining technique. Concept lattice is an ideal tool for such a task, hence we use the visualization power of concept lattice to generate clusters of functions within feature-specific functions and omnipresent functions. Alternatively, we apply the sequential pattern mining for the second time on the extracted execution patterns of the previous steps to separate feature-specific patterns from omnipresent patterns. This step is discussed in Section 5.5.

- *Structural evaluation*: in a further operation, by associating the functions of feature specific patterns, which implement the corresponding feature, to the system's structural modules, i.e., files of the system, two metrics for measuring *module cohesion* and *feature functional scattering* are obtained that together provide a means for measuring the impact of individual features on the structure of the software system.

1.3 Thesis Contribution

This thesis presents an approach in dynamic analysis of software systems to associate software functionalities to source code and as a byproduct provides a means for structural evaluation of software systems. The proposed approach takes advantage of dynamic analysis, data mining technique sequential pattern discovery, string processing algorithm repetition pattern finding, as well as the visualization power of the concept lattice analysis to provide comprehensive information about the software system from different aspects. The contributions of this thesis to the software maintenance field can be categorized as follows.

- Devised a novel pattern based approach to dynamic analysis of a software system that employs data mining techniques to extract valuable information out of noisy execution trace data.

- Proposed a technique to reduce the large sizes of the execution traces by eliminating the loop-based repetitions.

- Proposed a new technique for eliminating the sub-patterns that are generated along with the execution patterns.
- Identified the set of core functions that implement both specific features and common features of software systems.
- Provided a measure of scattering of the feature functionality to the structural modules as well as a measure of cohesion for a structural module.
- Visualized the functional distribution of specific features on a lattice using concept lattice analysis.

As a result of this research, we implemented a prototype tool as an Eclipse [3] plug-in using Java programming language. The implemented toolkit and the case studies are discussed in Chapter 7.

1.4 Limitations of the Technique

The presented approach in this thesis has some limitations as follows.

Limitations pertinent to the dynamic analysis approach: the proposed dynamic analysis is based on executing a group of feature-specific task scenarios on the program under study and observing the runtime executions; hence, the familiarity of the user with the application domain and the subject system is required. In addition, similar to any dynamic analysis technique the results of the proposed dynamic analysis indicate the properties of the input task scenarios rather than the properties of the entire system.

Limitations pertinent to the current implementation: there are many challenges in dynamic analysis of a software system that might restrict the applicability of the current implementation of the technique; among them, managing the huge sizes of the execution traces (tens of thousands of function calls in a medium size system), dealing with large number of extracted patterns from data mining operation, and identifying the real patterns from noise patterns, are notable.

1.5 Thesis Overview

The remaining chapters of this thesis are organized as belows.

Chapter 2: describes an overview of the related work in the area of dynamic analysis of the software system and feature to source code assignment.

Chapter 3: provides a detailed discussion of the formal definitions that are used throughout the thesis.

Chapter 4: presents a discussion of the techniques and algorithms that are employed throughout this study, including the sequential pattern mining algorithm, execution pattern post-processing algorithm, and concept lattice analysis techniques.

Chapter 5: presents the dynamic analysis approach for software feature to source code assignment. In this chapter, the steps of the proposed approach are explained in detail.

Chapter 6: provides the proposed structural evaluation technique.

Chapter 7: presents the results of the experimentations with the Xfig drawing tool and the Pine email system.

Chapter 8: provides a conclusion for the whole thesis and forms the basis for the future research.

Appendix: describes the implementation of the proposed prototype toolkit as an Eclipse plug-in.

Chapter 2

Related Work

In this section, we briefly present the approaches in dynamic analysis of a software system that relate to our works. In Section 2.1 we describe the approaches in software reverse engineering that employ data mining techniques. Section 2.2 elaborates on the existing approaches of application of concept lattice analysis in the software reverse engineering, and finally, in Section 2.3 we present recent approaches in dynamic analysis of software systems.

2.1 Data Mining

Fayyad et al. [16] defines data mining in databases as the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in the large databases. Data mining, in fact, aims at discovering unexpected, useful and simple patterns, and it is an inter-disciplinary research area. Recently, the application of data mining techniques in the software reverse engineering has been investigated. In static analysis of software systems, Montes and Carver [10] use association rule mining to identify subsystems in the database representation of the software system.

Sartipi [27] proposes a clustering method based on application of association rule mining where the similarity values between the system entities are defined based on the extracted association rules. In dynamic analysis of software systems, El-Ramly et al. [14] applied a sequential pattern mining technique to find interaction patterns between graphical user interface components. Their algorithm, so-called IPM, discovers frequently occurring patterns in program's interface snapshots. Consequently, an expert translates the extracted patterns to a use-case scenario. In [35] a web-mining technique is applied on program dynamic call graphs, where nodes represent classes and edges represent method invocation. In this approaches, classes (nodes) that depend on many other classes are identified using the web mining algorithm HITS. As a result, the classes in the software system that play an active role in the system are identified through this approach. In the presented approach in this thesis, we use data mining algorithm sequential pattern mining in order to extract frequent patterns of function calls. In this work, we utilize the "support" of each extracted pattern to filter out the noisy patterns.

2.2 Concept Lattice Analysis

Concept lattice analysis provides a way to identify groupings of *objects* that have common *attributes*. The mathematical foundation was laid by G. Birkhoff [7] in 1940. In 1993, work on the application of concept lattice analysis in the area of reverse engineering was initiated. Concept lattice analysis has been used for modularization of legacy code [28, 18, 31], where the relation between program functions and their attribute values (e.g., global variables, used types) are the basis for concept construction.

Recently, the application of concept lattice in dynamic analysis of software systems has been investigated. Eisenbarth et al. [12, 13] proposed a formal concept lattice analysis to locate computational units that implement a certain feature of the software system. They define a relation between task scenarios and program functions, where all the functions that are invoked during execution of a task scenario is considered as the attribute of that scenario. Similarly, we apply concept lattice analysis to the relation between specific feature in a scenario and certain program functions invoked during the scenario execution. However, we filter out noise functions by applying sequential pattern mining which has a huge effect on reducing the complexity and increasing the understandability of the concept lattice. Tonellan et al. [30] applied concept lattice analysis on execution traces of a software system to mine the potential program-aspects that exist in the software.

Concept lattice analysis and data mining techniques both extract maximal sequences of execution traces that contain important information to be analyzed. However, sequential pattern analysis has the control over the number of generated common traces using the minimum support. In the proposed technique we amalgam the advantages of both techniques to explore the non-trivial execution patterns as a means to explore the functionality of a specific software feature.

2.3 Dynamic Analysis

In [6, 15], Bell and Ernst studied the characteristics of dynamic analysis of software systems and compared the properties of dynamic analysis technique with those of a static analysis. A typical approach to dynamic analysis of a software systems is based on executing a set of task scenarios on the software system and analyzing the corresponding execution traces. In an approach to software understanding using execution traces Pauw et al. [21] visualized the execution traces of object-oriented programs and provided a set of navigational and analytical techniques to facilitate the execution trace exploration in various abstraction levels. Fischer et al. [19] used execution traces as clues for tracing the evolution of a software system. In [36] a heuristic exploration to execution traces has been proposed that aims at clustering the program functions based on their invocation frequency. Execution traces are also used in performance analysis of software systems. In [20, 33, 8] performance analysis of parallel system is studied by using execution traces of the software systems. In [8, 33] a program's execution trace is searched for certain predefined patterns that indicate inefficient behavior. In [11] a time interval analysis is applied to the execution traces to locate components that implement a certain feature in a distributed application. Traces of execution within the intervals with and without a specific feature being active are compared to locate the code component that implement that specific feature. Although this method is quite interesting, but since activation of a feature might be interleaved with other functionalities of the software, determining an exact time interval for activation of a specific feature is not always feasible.

N. Wilde et al. [32] proposed a set difference approach for locating software features in the source code; where the set of functions in the related scenario executions (those that execute a specific feature) are differentiated from scenario executions that do not invoke that specific feature in order to extract the specific feature's functionality. In our approach, we also use the notion of feature specific scenarios, however we extract patterns of execution traces as evidences of the feature functionality.

In contrast to the above techniques, our approach exploits a novel analysis technique to handle large sizes of the execution traces, and allows an intuitive and promising process of feature to component allocation that consequently leads us to measure the functional scattering and cohesiveness of the software structural units.

Chapter 3

Formal Definitions

In this chapter, we define the common terminology that we use throughout this thesis to describe the execution pattern mining and pattern analysis aspects of the proposed approach. We provide a model for representing the task scenarios in Section 3.1. In Section 3.2 a representation for a software execution using dynamic call tree is provided. In Section 3.3 we present the definitions for execution pattern mining. Finally, Section 3.4 uses the definitions presented in this chapter in order to model the feature to source code assignment problem.

3.1 Scenario Model

In the context of this work, we model a scenario as a sequence of relevant features of the software system. In this way, each software feature is considered as the building blocks of the task scenarios.

- *feature* ϕ is a unit of software requirement that describes a single functionality in the software system under study. Φ is the set of all features in the system.

- *scenario* s is a sequence of features $\phi \in \Phi$; thus $s = [\phi_1, \phi_2, \ldots, \phi_n]$. Also $\mathcal{S}$ is the set of all applicable scenarios on the system.

- *feature-specific scenario set* S_ϕ is a set of scenarios that share specific feature ϕ; thus[1]

$$S_\phi = \{s \mid s \in \mathcal{S} \ \wedge \ \exists\, \phi' \in s \bullet \ \phi' = \phi\}.$$

3.2 Software System Model

Based on the static dependencies that exist in the source code of a software system, one can model the software system with a call graph, where nodes represent functions and edges represent function calls. In this representation each scenario execution on the software system corresponds to a traverse on the system call graph. In order to formalize the dynamic aspects of the software system in this work, we represent this source graph traversal with a *dynamic call tree.* In this representation, two different invocations of a single function are represented with two different nodes and edges of the tree are representing the function calls.

- Let $\mathcal{F}$ be the set of all function names in the subject software system.

- Let $\mathcal{F}'$ be the set of all invocations (calls) of functions $f \in \mathcal{F}$. In this context, two different invocations of a single function $f \in \mathcal{F}$ are represented as f^i and f^j $(i \neq j)$.

- *Dynamic Call Tree* (*DCT*) is a tree which represents the execution of a scenario on the software system. In this representation, nodes represent functions and

[1]We use set membership operator $\in$ as a sequence membership operator as well.

edges represent function calls. $DCT =< \mathcal{F}', E >$, where E is a set of ordered pairs [2]such that: $E \subset \mathcal{F}' \times \mathcal{F}'$

- Dynamic call tree *preprocessor* Π is a tree pruning operation which removes multiple instances of identical subtrees in a dynamic call tree that are repeated under a particular parent node. $\Pi : DCT \rightarrow DCT$

In this work, dynamic call trees are obtained from execution of task scenarios on the instrumented software system. We model a software system as a set of all possible dynamic call trees that each corresponds to one task scenario execution. We also model a scenario execution as a look up operation which returns the corresponding dynamic call tree of a scenario in the software system.

- Let *software system* Ψ be the set of all possible dynamic call trees.

- Let *scenario execution* $\mathcal{E}(s)$ on software system Ψ be a look up function which returns the corresponding dynamic call tree of scenario s. $\mathcal{E} : \mathcal{S} \rightarrow \Psi$

We transform a dynamic call tree to an *execution trace* for further analysis in this work. Each execution trace is represented with a sequence of function names. In this formalism, execution traces are built by the depth first traversal of the dynamic call trees.

- *Execution Trace* $\mathcal{T}$ is a sequence of function names from $\mathcal{F}$.

- A *dct* $\in DCT$ is mapped to an execution trace $t \in \mathcal{T}$ using a depth first traversal DFT on the *dct*, where the sequence of visited nodes in this traversal constitute execution trace t. $DFT : DCT \rightarrow \mathcal{T}$

[2]Note that E preserves the required constraints of a tree.

3.3 Execution Pattern

In this work, we define an *execution pattern* as a contiguous part of an execution trace that exists in certain number of execution traces, namely the support of the pattern.

- Let *repository* R_{S_ϕ} be the set of all extracted execution traces according to the execution of task scenarios in feature-specific scenario set S_ϕ. Thus we would have:
  $R_{S_\phi} = DFT(\Pi(\mathcal{E}(S_\phi)))$

- An execution pattern $p \in \mathcal{T}$ is defined as a contiguous sequence of functions $f \in \mathcal{F}$ that is supported by at least $MinSupport$ number of execution traces in the repository R_{S_ϕ}.

- An execution trace t *supports* execution pattern p iff p is a subsequence of t, such that: $\exists i\ \forall j\ (0 \leq i \ \ \wedge \ \ i \leq j < (i + |p|) \ \ \rightarrow \ \ p[j - i] = t[j])$.

- Let *support set* of pattern p be the set of all execution traces that support execution pattern p.

- An *execution pattern miner* $\Upsilon_n(R_{S_\phi})$ is a function which extracts all execution patterns that are supported by at least $n\%$ of execution traces in R_{S_ϕ}.
  $\Upsilon_n : \ \ Powerset(\mathcal{T}) \rightarrow \ \ Powerset(\mathcal{T})$

3.4 Feature to Source Code Assignment

Depending on the level that functions are participating in execution patterns of different feature-specific scenario sets, we define two categories of functions: *feature-specific* functions and *omnipresent* functions.

- Function f is *associated* with feature-specific scenario set S_ϕ such that: $\exists p \in \Upsilon_n(R_{S_\phi}) \bullet f \in p$.
- A function is categorized as an omnipresent function iff it is associated with almost every feature-specific scenario set.
- A function f is a feature-specific function for feature ϕ iff f is associated with only the unique feature-specific scenario set S_ϕ.

In this context, the group of all feature-specific functions for feature ϕ constitute the mapping of feature ϕ to the software system source code.

Chapter 4

Techniques

In this chapter, we discuss the major techniques that are used throughout this thesis. We briefly present the application of sequential pattern mining in Section 4.1 and mathematical concept lattice analysis in Section 4.2. The former is used to extract highly repeated execution patterns as a result of applying sequential pattern mining on the pruned execution traces. The later is applied on the extracted execution patterns in order to cluster the functions that exist within common / feature-specific patterns.

4.1 Execution Pattern Mining

In this section, we describe the application of a data mining technique to discover sequences of functions in a software system that correspond to certain system features. In the data mining literature, sequential pattern mining is used to extract frequently occurring patterns among the sequences of customer transactions [5]. In this context, the sequence of all transactions corresponding to a certain customer (already ordered by increasing transaction-time) is referred to as a *customer-sequence.* A customer-

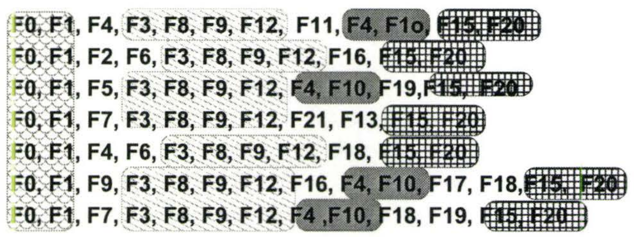

F0, F1, F4, F3, F8, F9, F12, F11, F4, F1o, F15, F20
F0, F1, F2, F6, F3, F8, F9, F12, F16, F15, F20
F0, F1, F5, F3, F8, F9, F12, F4, F10, F19, F15, F20
F0, F1, F7, F3, F8, F9, F12, F21, F13, F15, F20
F0, F1, F4, F6, F3, F8, F9, F12, F18, F15, F20
F0, F1, F9, F3, F8, F9, F12, F16, F4, F10, F17, F18, F15, F20
F0, F1, F7, F3, F8, F9, F12, F4 ,F10, F18, F19, F15, F20

Figure 4.1: An execution trace repository containing 7 execution traces. The four shaded areas correspond to four execution patterns with minimum support 3.

sequence *supports* a sequence *s* if *s* is a sub-sequence of this customer-sequence. A frequently occurring sequence of transactions (namely a pattern) is a sequence that is supported by a user-specified minimum number of customer-sequences known as the *minimum support* of this pattern, namely *MinSupport* the pattern.

4.1.1 Execution Pattern

In this study we use a modified version of the sequential pattern mining algorithm by Agrawal [5], where an execution pattern is defined as a contiguous part of an execution trace that is supported by *MinSupport* number of execution traces. In this analysis we use an execution trace as a customer-sequence defined above. The formal definition of an execution pattern has been provided in Section 3.3. In this formalism, each execution pattern is associated with a set of feature-specific task scenarios and reveals the common functionality that is invoked within these scenarios.

In Figure 4.1 an example of an execution trace repository and its corresponding execution patterns is shown. In this example the *MinSupport* is 3.

A typical sequential pattern mining algorithm allows extracting noncontiguous sequences of function calls. In most cases, this characteristic drastically increases the time/space complexity of the pattern mining algorithm and will complicate the dynamic analysis. In the presented approach, each extracted sequential pattern is a contiguous sequence of function calls that exists in different execution traces. This strategy produces meaningful execution patterns that correspond to core functions implementing specific functionalities of the system. Whereas, as we got from our experiments, extracting execution patterns that contain noncontiguous function invocations would generate an overwhelming number of meaningless patterns that consist of unrelated parts of the execution traces.

4.1.2 Algorithm

In the following an overview of the proposed execution pattern mining algorithm is provided. This algorithm consists of two main procedures: candidate two-items pattern generation (Procedure *cpGenerator*) and pattern extension (Procedure *DoExtend*). Procedure *cpGenerator* accepts the repository R_{S_ϕ} as input and simply generates all two-items patterns. Among the generated two-items patterns those that meet the $MinSupport$ constraint are stored in the *candidate pattern repository.*

Procedure *DoExtend* increases the length of the patterns of the pattern repository iteratively. This procedure uses the operation *extend* to extend the patterns. In the following an overview of the operation *extend* is provided:

> A pattern p can be extended by a candidate pattern cp if p ends exactly where cp starts. The resulting extended pattern p' is constructed by concatenating p and cp. The support set of this pattern consists of traces in the intersection of support sets of cp and p that also support p'.

The pattern extension stage starts with storing all candidate patterns in a pattern repository (see Procedure *DoExtend*). This procedure iterates as long as any pattern can be extended. In each iteration, for each pattern p in the pattern repository, it checks if p can be extended using candidate patterns in the candidate pattern

Procedure `cpGenerator`

```
Input: Set $R_{S_\phi}$
Result: Set $CPR$ //CPR is the candidate pattern repository.
Variable: MultiSet $\mathcal{MS}$ //A multiset is a set for which repeated elements are
considered. begin
    $CPR \leftarrow$ empty set;   $\mathcal{MS} \leftarrow$ empty set;
    foreach $t \in R_{S_\phi}$ do
        // t is an execution trace;
        for $i \leftarrow 0$ to $|t| - 2$ do
            add $t[i..i+1]$ to $\mathcal{MS}$;
        end
    end
    foreach $ms \in \mathcal{MS}$ do
        if Multiplicity($ms$) $\geq MinSupport$ then
            add $ms$ to $CPR$;

    end
end
```

repository. If p can not be extended in an iteration then it is stored as an execution pattern. When no more patterns can be extended in an iteration Procedure *DoExtend* terminates.

One drawback of the mentioned execution pattern mining algorithm is that it generates certain sub-subsequences of a final execution pattern, that drastically increases the number of generated execution patterns. Note that the pattern extend operation extends pattern p from end of the p. In this case, all sub-sequences of pattern p that terminate at end of the p may be generated along with p.

Suppose the following situation:

$$p_1 = \{2,3,4\} \quad cp = \{4,5\} \quad p'_1 = \{2,3,4,5\}$$

$$p_2 = \{1,2,3,4\} \quad cp = \{4,5\} \quad p'_2 = \{1,2,3,4,5\}$$

in this case pattern p'_1 would not grow up more, whereas pattern p_2 grows and becomes a super-sequence for pattern p'_1.

Procedure `DoExtend`

```
Input: Set $CPR$ //$CPR$ is the candidate pattern repository.
Result: Set $EPR$ //$EPR$ is the resulting execution pattern repository.
Variable: Set $PR$ //$PR$ corresponds to the set of growing patterns.
Variable: Set $Temp$
begin
    $EPR \leftarrow$ empty set;
    $PR \leftarrow CPR$;
    while $PR$ has an element do
        $Temp \leftarrow$ empty set;
        foreach $p \in PR$ do
            $extendedOnce \leftarrow FALSE$;
            foreach $c \in CPR$ do
                $p' \leftarrow p + c$    //operator + denotes to operation pattern extend.;
                if $support(p') \geq MinSupport$ then
                    $extendedOnce \leftarrow TRUE$;
                    add $p'$ to $Temp$;

            end
            if $extendedOnce = FALSE$ then
                add $p$ to $EPR$;

        end
        $PR \leftarrow Temp$
    end
end
```

```
TrieNode{
            String      functionName;
            Mark        mark;
            TrieNode    parent;
}
```

Figure 4.2: Data structure that is used to represent a Trie node.

We apply a novel sub-pattern elimination operation that has a major impact on enhancing the pattern analysis performance. The sub-pattern elimination operation is discussed in more details in the following discussion.

4.1.3 Sub-Pattern Elimination

In order to identify and eliminate sub-patterns of a final execution pattern, we use a *Trie* data structure and annotate its nodes with the function names. A Trie is a tree data structure that stores the information about the contents of each node in the path from the root to the node, rather than the node itself. In Figure 4.2 the data structure that we used for representing the tree nodes in Trie data structure is shown. Each *TrieNode* has an enumerated type "Mark" that can have values "*final*" or "*subPattern*", where "Mark" is used to eliminate the sub-patterns.

In doing so, the sequence of functions in each execution pattern p is stored along a path from the root to the leaf of the Trie, and the corresponding leaf is marked as *final* if it does not already exist in the Trie. In this setting, all sub-sequences of p that terminate at the end of p are inserted in the Trie as well. The leaf nodes that correspond to these paths are marked as *subPattern*. Procedure *TrieInsert* illustrates an overview of the above mentioned operation.

Figure 4.3(a) depicts the Trie data structure after inserting final execution patterns $p_1 = \{F3, F8, F9, F12\}$ and $p_2 = \{F8, F9, F13\}$.

Procedure `TrieInsert`

```
Input: Pattern $P$
Global Variable: $Trie\ trie$;
Result: Inserting pattern $P$ along with all its sub-patterns that terminate at
        end of $P$ to the $trie$ data structure.
begin
    $start \longleftarrow 0$;   $mark \longleftarrow "final"$;
    while $start < |P|$ do
        TrieNode $t \longleftarrow trie.root$;
        $index \longleftarrow start$;
        while $index < |P|$ and $t.hasChild(P[index])$ do
            $t \longleftarrow t.getChild(P[index])$;
            $index = index + 1$;
        end
        // check to see if P is already in Trie.;
        if $index \equiv |P|$ then
            // change node's mark only from "final" to "subPattern";
            if $mark \equiv "subPattern"$ then
                Mark $t$ as "subPattern";
            exit;
        // add the remainder of the input pattern to the Trie;
        while $index < |P|$ do
            //add a child to the t and return the newly added child;
            $t \longleftarrow t.addChild(P[index])$;
            $index = index + 1$;
        end
        Mark $t$ as $mark$;
        // add all subsequences of the input pattern to the Trie;
        $start = start + 1$;
        $mark \longleftarrow "subPattern"$;
    end
end
```

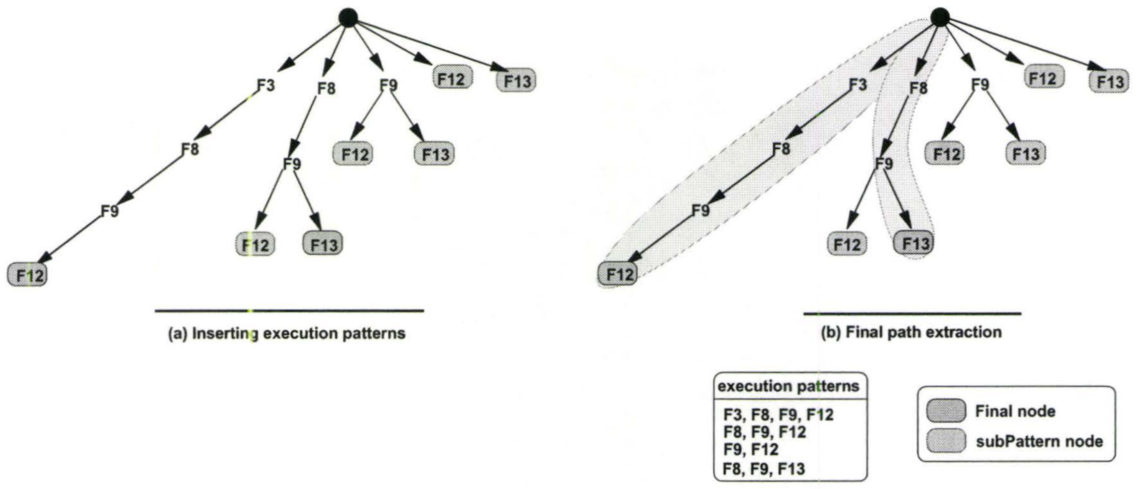


Figure 4.3: SubPattern elimination: (a) inserting execution patterns in Trie and marking leaves as final and subPattern (b) final execution pattern extraction, shaded areas correspond to final paths.

We call a path that starts from an arbitrary node of the Trie and ends at a leaf that is marked as *final*, a *final path*. Procedure *TrieExtract* illustrates an overview of the operation of extracting all final paths that start from node t. This procedure is a simple depth first traversal on the Trie that stores the visited nodes in a stack, thus at any node t' the nodes in the stack represent a path from t to t'. By extracting all final paths that start from the root of the Trie, we will generate all final patterns. Figure 4.3(b) depicts the final paths, as two shaded areas, corresponding to final execution patterns $p = \{F3, F8, F9, F12\}$ and $p_2 = \{F8, F9, F13\}$.

4.2 Concept Lattice Analysis

Mathematical concept analysis was first introduced by Birkhoff in 1940 [7]. In this formalism, a binary relation between a set of "objects" and a set of "attribute-values" is represented as a lattice. A *concept* is a maximal collection of objects sharing maximal common attribute-values. A *concept lattice* can be composed to provide

```
Procedure TrieExtract
  Input: TrieNode $t$
  Result: Extracting all final paths starting from $t$.
  Data: Stack $stack$
  begin
      $stack.push(t)$;
      if $t$ is a Leaf then
          if $t$ is marked as "final" then
              { $stack$ contains one final path now};

      else
          forall children of $t$ do
              Call TrieExtract;
          end
      end
      $stack.pop()$;
  end
```

significant insight into the structure of a relation between objects and attribute-values such that each node of the lattice represents a concept. In a binary relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$ between a set of *objects* $\mathcal{O}$ and their *attributes* $\mathcal{A}$, the triple $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ is called a *formal context*. For any set of objects $O \subset \mathcal{O}$, we define *shared attributes* $\sigma(O)$ as the set of attributes that are shared among objects in O.

$$\sigma(O) = \{a \in \mathcal{A} | \forall o \in O \bullet (o, a) \in \mathcal{R}\}$$

Similarly, for any set of attributes $A \subset \mathcal{A}$, the set of common objects is defined as *shared objects* $\tau(A)$.

$$\tau(A) = \{o \in \mathcal{O} | \forall a \in A \bullet (o, a) \in \mathcal{R}\}$$

A formal context can be visualized with a relation table, where each row represents an object and each column represents an attribute. An object o_i and an attribute a_j are in the relation (i.e., object o_i has attribute a_j) iff the cell at row i and column j is marked in the relation table. In Table 4.1 an example of a formal context is provided.

	f1	f2	f3	f4	f5
s1	×	×			×
s2	×	×		×	
s3	×		×		

Table 4.1: An example of a relation table with 3 objects and 5 attributes.

The following equations hold for this context table:

$$\sigma(\{s1, s2\}) = \{f1, f2\}$$

$$\tau(\{f1\}) = \{s1, s2, s3\}$$

A *concept* c is defined as a pair $c =< O, A >$ such that:

$$O = \tau(A) \quad \wedge \quad A = \sigma(O)$$

where O is called the *extent* of c, denoted by $Ext(c)$, and A is called the *intent* of c, denoted by $Int(c)$. Such a concept corresponds to a maximal rectangle in its context table. Table 4.2 presents all concepts of the relation table in Table 4.1.

The *infimum* of two concepts is computed by joining their intents and intersecting their extents.

$$c_1 \wedge c_2 \quad = \quad < Ext(c_1) \cap Ext(c_2), \sigma(\tau(Int(c_1) \cup Int(c_2))) >$$

The infimum describes a set of common attributes of two sets of objects. Therefore the infimum of two concepts can be rewritten as:

$$c_1 \wedge c_2 \quad = \quad < Ext(c_1) \cap Ext(c_2), \sigma(Ext(c_1) \cap Ext(c_2)) >$$

The *supremum* is computed by joining the extends and intersecting the intents of two concepts:

$$c_1 \vee c_2 \quad = \quad < \tau(\sigma(Ext(c_1) \cup Ext(c_2))), Int(c_1) \cap Int(c_2) >$$

c_x	$< Ext(c_x),\ Int(c_x) >$
c_1	$< \{s1, s2, s3\},\ \{f1\} >$
c_2	$< \{s1, s2\},\ \{f1, f2\} >$
c_3	$< \{s1\},\ \{f1, f2, f5\} >$
c_4	$< \{s2\},\ \{f1, f2, f4\} >$
c_5	$< \{s3\},\ \{f1, f3\} >$

Table 4.2: Concepts of the context table in Table 4.1.

The supremum describes the set of objects that share all the attributes in the intersection of two sets of attributes. Hence the supremum of two concepts can be represented as:

$$c_1 \vee c_2 \;=\; < \tau(Int(c_1) \cap Int(c_2)), Int(c_1) \cap Int(c_2) >$$

A concept lattice is an acyclic directed graph where nodes represent concepts and edges represent subconcept relations. A concept (O_0, A_0) is a *subconcept* of concept (O_1, A_1), if $O_0 \subset O_1$. This relationship defines a complete partial order over the set of all concepts of a given formal context $\mathcal{C}$, that can be represented as a *concept lattice* $\mathcal{L}(\mathcal{C})$.

Complete information of each concept c (i.e. node) in the concept lattice is provided by the pair $< Ext(c), Int(c) >$. However, the same information can be represented in a more concise form by marking a concept c with an attribute $a \in Int(c)$ or with an object $o \in Ext(c)$. The unique node in the concept lattice that is marked by attribute a is computed by function $\mu(a)$ as follows:

$$\mu(a) = \bigvee\{c \in \mathcal{L}(\mathcal{C}) | a \in Int(c)\}$$

in doing so, each attribute a will label the most general concept that has a in its intent. As a result, those attributes that are shared among most of the objects will appear in the upper region of the lattice, and those that are more specific label the concepts in the lower region of the lattice.

The unique node that is marked by object o is:

$$\gamma(o) = \bigwedge\{c \in \mathcal{L}(\mathcal{C}) | o \in Ext(c)\}$$

analogously, o will label the most specific concept that has o in its extent.

Visualizing the concept lattice following the above mentioned labeling mechanism provides certain characteristics for the lattice as follows.

- Each lattice node (i.e., a concept) might be labeled with objects and attributes.
- Every object has all attributes that are defined at that node or above it in the lattice (directly above or separated by some links).
- Every attribute exists in all objects that are defined at that node and nodes in the sub-lattice below it (directly below or separated by some links).

A concept lattice can be used to collect the set of shared attributes contained in a set of objects such that the shared attributes appear in the nodes that are located in the upper region of the lattice. Consequently, the nodes in the lower region of the lattice collect the attributes that are specific to the individual objects in that region. In Section 5.5, we exploit this property to cluster the functions of the extracted execution patterns.

Figure 4.4 depicts the corresponding concept lattice of the formal context of Table 4.1. In this lattice, node with label f_2 represents the concept $c2 =< \{s_1, s_2\}, \{f_1, f_2\} >$. Note that $Ext(c_2)$ is obtained by collecting the objects that are shown in the nodes below it. Similarly, the intent of c_2 contains all the attributes that are labeled the nodes above it.

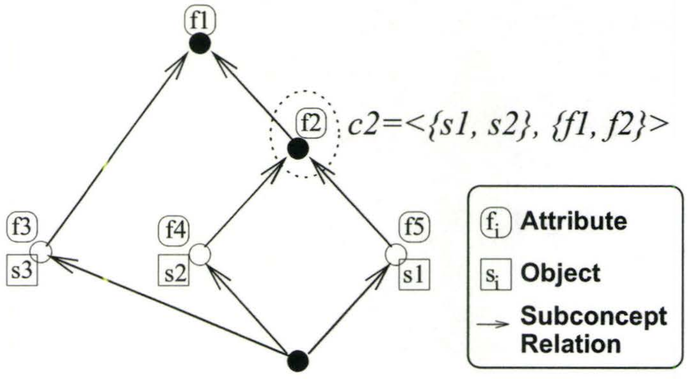


Figure 4.4: Corresponding concept lattice of Table 4.1

Chapter 5

Dynamic Analysis

A typical approach to dynamic analysis deals with extracting software execution traces corresponding to a set of carefully selected task scenarios and reveals the realization of the scenarios' functionalities within the software system components. In this thesis, we propose an approach to dynamic analysis of software systems based on the frequently appearing patterns in execution traces, in order to identify the implementation of the software features in the source code. We execute a set of task scenarios with a specific shared feature, referred to as feature-specific scenario set S_ϕ, on the software system in order to generate execution traces. The application of a sequential pattern mining algorithm on the extracted execution traces allows us to highlight the feature related system functionality. Based on the proposed framework in Chapter 1 that is repeated below and the definitions of Chapter 3, we define this process with the following steps.

1. Define feature-specific scenario set S_ϕ.

2. Execute the scenarios in S_ϕ on the subject software system and generate the corresponding dynamic call trees such that DCT_{S_ϕ} is the set of all dynamic call

trees for the scenario set S_ϕ. Each dynamic call tree represents an unpruned call trace after execution of the software system:

$DCT_{S_\phi} = \mathcal{E}(S_\phi)$

where $\mathcal{E}$ is the scenario execution operation.

3. Preprocess the extracted dynamic call trees DCT_{S_ϕ} in order to eliminate the loop-based repetitions and extract the corresponding execution traces R_{S_ϕ}:

   $R_{S_\phi} = DFT(\Pi(DCT_{S_\phi}))$

   where Π represents preprocessing and DFT represents the depth first traversal operations.

4. Apply the execution pattern mining Υ_n described in Section 4.1 and extract the set of execution patterns P_{S_ϕ} that exist in R_{S_ϕ}.

   $P_{S_\phi} = \Upsilon_n(R_{S_\phi})$

We apply the above process on different features of the software system and extract groups of execution patterns that each reflect the software functionality corresponding to the experimented features. In this context, a post-processing of the generated execution patterns will allow us to extract patterns that exclusively correspond to a single feature-specific scenario set from those that are shared between all feature-specific scenario sets. This chapter is structured as follows. An overview of the program instrumentation is presented in Section 5.1. In Section 5.2 we discuss the feature-specific scenario set selection. A detailed discussion of execution trace generation is provided in Section 5.3. Finally in Section 5.4, we will discuss the the execution pattern extraction and execution pattern analysis.

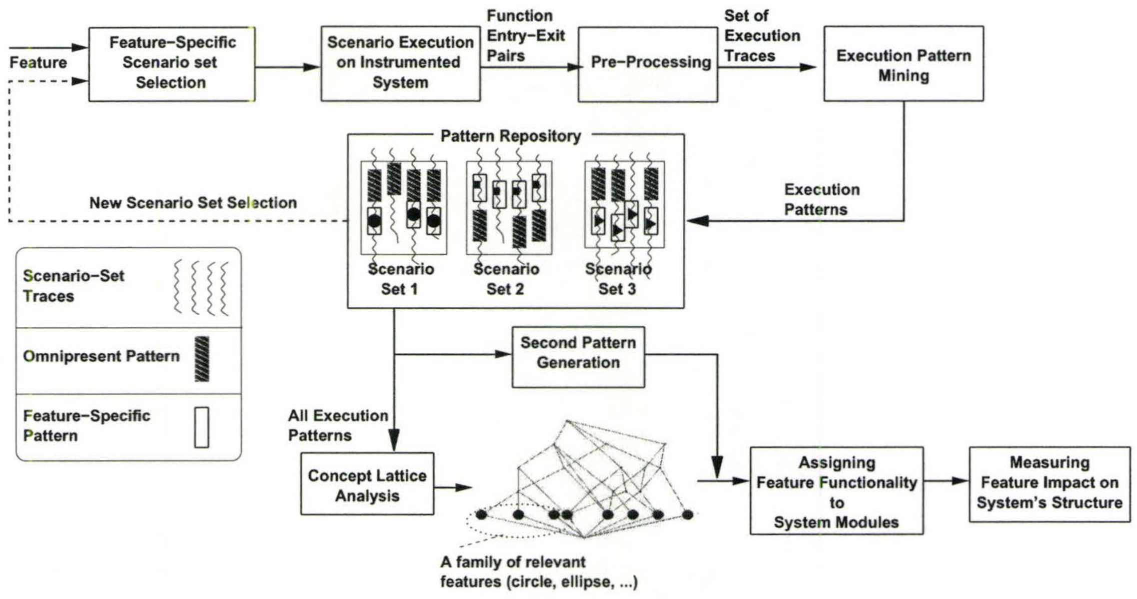
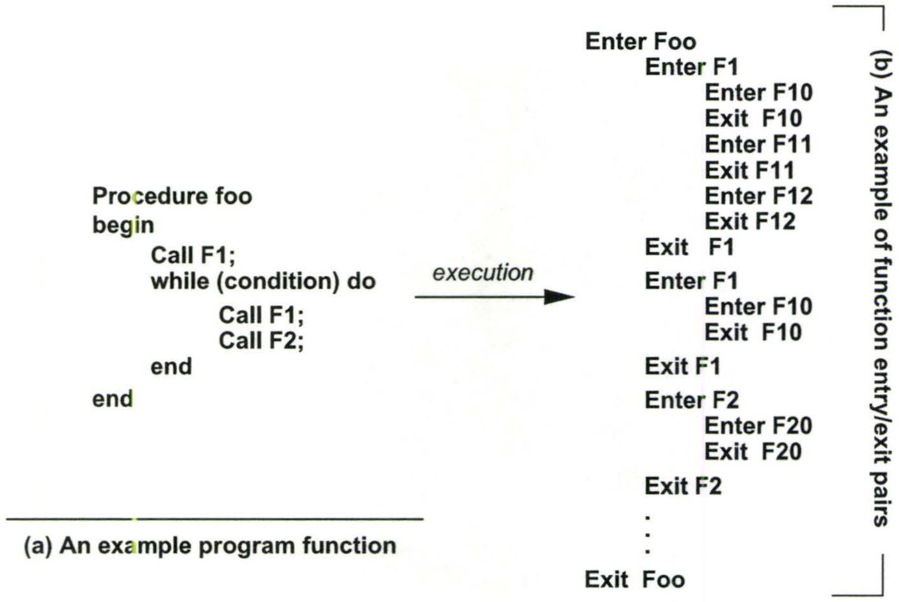

Feature
Feature-Specific Scenario set Selection
Scenario Execution on Instrumented System
Function Entry-Exit Pairs
Pre-Processing
Set of Execution Traces
Execution Pattern Mining
Pattern Repository
New Scenario Set Selection
Execution Patterns
Scenario Set 1
Scenario Set 2
Scenario Set 3
Scenario-Set Traces
Omnipresent Pattern
Feature-Specific Pattern
Second Pattern Generation
All Execution Patterns
Concept Lattice Analysis
Assigning Feature Functionality to System Modules
Measuring Feature Impact on System's Structure
A family of relevant features (circle, ellipse, ...)

Figure 5.1: Proposed framework for identifying the implementation of the functional aspects of a software system in the source code as a means to incorporate semantics into static analysis techniques.

5.1 Instrumentation

Instrumentation refers to the process of inserting particular pieces of code in the subject software system in order to acquire specific information about the execution of the software system. Instrumentation can be performed both at the source code and at the binary image level. In the proposed approach, we adopt *Aprobe* [29] which is a binary level software instrumentation tool. Aprobe inserts patches, namely *probes*, within the binary image of the executable program. In this work, we used a pre-defined *probe* (namely *trace*) that generates text messages at both entrance and exit points of each function. Therefore, by executing the subject software system a *function entry/exit pairs* is obtained that is represented as a dynamic call tree. In Figure 5.2(a) an example function is shown. Extracted function entry/exit pairs for a sample execution of this function are shown in Figure 5.2(b).



```
Procedure foo
begin
    Call F1;
    while (condition) do
        Call F1;
        Call F2;
    end
end
```

```
Enter Foo
    Enter F1
        Enter F10
        Exit  F10
        Enter F11
        Exit F11
        Enter F12
        Exit F12
    Exit  F1
    Enter F1
        Enter F10
        Exit  F10
    Exit F1
    Enter F2
        Enter F20
        Exit  F20
    Exit F2
    .
    .
    .
Exit  Foo
```

execution
(a) An example program function
(b) An example of function entry/exit pairs

Figure 5.2: Instrumentation: (a) sample function *foo()*, (b) extracted function entry/exit pairs for a sample execution of function *foo()*.

5.2 Scenario Selection

Important features of the subject software system are identified by investigating the system's user manual, on-line help, similar systems in the corresponding application domain, and also user's familiarity with the system. For each particular feature ϕ we select a set of relevant task scenarios where feature ϕ is shared among all scenarios. We call this set of scenarios a feature-specific scenario set. For example, in the case of a drawing tool software system such as Xfig, a group of scenarios that share the operation "move" to relocate a drawn figure on the computer screen would constitute such a feature-specific scenario set. Bellow a set of five feature-specific scenarios for the operation "move" on Xfig drawing tool is presented:

start, draw rectangle, *move*, exit

start, draw ellipse, *move*, exit

start, draw circle, *move*, exit

start, draw arc, *move*, exit

start, draw polygon, *move*, exit

By executing the scenarios of the feature-specific scenario set S_ϕ on the instrumented software system, a group of function entry/exit pairs (dynamic call trees) is extracted that should be preprocessed and converted to execution traces for the further analysis.

5.3 Execution Trace Generation

In this section, we discuss the steps for generating execution traces. We start with a detailed discussion about the rationals for execution traces preprocessing. Then we describe the preprocessing mechanism which includes dynamic call tree generation and dynamic call tree pruning.

5.3.1 Preprocessing

Dynamic analysis of a medium size software system using execution traces can produce very large traces ranging to thousands or tens of thousands of function calls. This would be a main source of difficulty in a typical dynamic analysis technique. The effective trace of functions for the intended scenario is cluttered by a large number of function calls from operating system, initialization and termination operations, utilities, repetition of sequences caused by the loops, and also noise functions that are interleaved within a sequence. Thus, prior to using the extracted function entry/exit pairs in further steps, the redundancies in the trace that are produced by program loops and recursive function calls should be eliminated. For our current state of analysis in this work, we ignore recursive function traces and focus on pruning the loop-based redundancies.

In this operation, we transform the function entry/exit pairs that is generated by instrumenting the software system into a dynamic call tree (Section 3.2), where

$$\begin{array}{l}
S := \textit{'Enter'}\ \ ID\ \ S\ \ \textit{'Exit'}\ \ ID\ \ S\ \mid\ \epsilon \\
ID := \ \ Letter\ \ (\ Letter\ \mid\ Digit\ \mid\ \textit{'\_'}\ )^* \\
Letter := [\ [\textit{'a'}\ ..\ \textit{'z'}\ ]\ +\ [\ \textit{'A'}\ ..\ \textit{'Z'}\ ]\ ] \\
Digit := [\ \textit{'0'}\ ..\ \textit{'9'}\ ]
\end{array}$$

Figure 5.3: Grammar for parsing the Aprobe instrumentation data.

nodes represent functions and edges represent function calls. Since each loop resides in the body of a function, the loops will form identical subtrees as the children of the parent function. In this context, the loop redundancy removal problem is reduced to identification of identical subtrees that are repeated under a particular node. In Procedure Foo a piece of code that produces a long trace with repetitions of "*F1, F2*" is shown. The following subsections elaborate the dynamic call tree generation and dynamic call tree pruning, respectively.

Procedure Foo. A dummy procedure which generates loop-based repetitions.

```
begin
    Call F1;
    while condition do
        Call F1;
        Call F2;
    end
end
```

5.3.2 Dynamic Call Tree Generation

The output of the software instrumentation using Aprobe (function entry/exit pairs) can be transformed into a dynamic call tree of the running program. In Figure 5.3, we present the context free grammar that we use for parsing the function entry/exit pairs.

In Figure 5.4 the data structure that we used for representing the dynamic call

```
GraphNode{
            String      name;
            Integer     ID;
            GraphNode   parent;
}
```

Figure 5.4: Data structure that is used to represent a graph node.

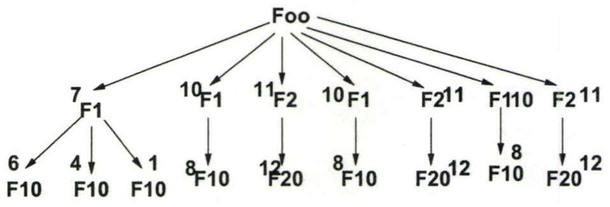


Figure 5.5: A dynamic call tree that is generated for an example execution of Procedure *Foo* in Figure 5.2.

tree nodes is shown. Each *GraphNode* in addition to its name and its parent has an integer *ID*. In the following, we give an overview of the proposed parser, namely *callTreeParser*, that parses the input function entry/exit pairs and generates the corresponding dynamic call tree. Procedure *callTreeParser* also assigns an integer *ID* to each tree node, where roots of identical subtrees have identical *ID*s. This procedure uses two auxiliary functions, *NextToken* and *GetNextToken*, where *NextToken* returns true if the lexical analyzer has another token in its input and *GetNextToken* finds and returns the next complete token from the lexical analyzer. One main duty of *callTreeParser* is to find the identical subtrees and to tag them with identical integer *ID*s. For this operation, we use a hash-table implementation which uses the *ID* values of the input *GraphNode*'s children to generate the output *ID* value. This mapping is done by a call to *idRepository.getID*.

Figure 5.5 illustrates a small portion of a dynamic call tree which is generated for an example execution of Procedure *Foo* in Figure 5.2. Each node in this call tree is also annotated with its *ID*. Note that function *F1* is called by function *Foo* several

Procedure `callTreeParser`

```
Result: This procedure parses the input function entry/exit pairs and
        generates the dynamic call tree.
// Before the first call to this procedure, we set variable currentNode to an instance of
GraphNode, namely root of the tree;
begin
    while NextToken() = 'Enter' do
        getNextToken();
        GraphNode newChild ⟵ new GraphNode;
        add newChild to the children of the currentNode;
        currentNode ⟵ newChild;
        callTreeParser();
        if NextToken() $\neq$ 'Exit' then
            Exit on Error;
        getNextToken();
        ID ⟵ idRepository.getID(newChild);
        label newChild with ID;
        currentNode ⟵ currentNode.getParent();
    end
end
```

times, however it acquires different *ID*s depending on its runtime behavior.

5.3.3 Dynamic Call Tree Pruning

In this section, we present an implementation for the dynamic call tree preprocessor $\Pi(dct)$ described in Section 3.2. As mentioned in previous subsection, we label each subtree with a unique integer *ID* where identical subtrees possess identical *ID*s, which has a great significance in localizing the loop-based redundancy elimination at the proper children of each node in the dynamic call tree. The dynamic call tree preprocessor intends to remove the multiple instances of identical subtrees that are repeated as the children of a particular node. In this operation, we first generate a string representation of *ID* values of these sibling subtrees. Then by applying a repetitive string finder algorithm (*Crochemore* [9]) we transform the original string (with repetitions) in the form of a new string with no repetitions. In this new string, each group of repetitions is shown as one instance of the repetition that is labeled with the number of the repetitions. For example, in Figure 5.6(a) the string *F1,F2,F1,F2, ..., F1, F2* is transformed into *(F1,F2)*^{n} in Figure 5.6(b). There may exist more than one pattern of repetitions for a given string and hence we apply the following heuristic in order to select the dominant pattern.

> The repetitive pattern with the highest power generates a pattern that is resulted from a program loop.

As a result, we keep subtrees that correspond to a single instance of each loop, which greatly reduces the complexity of the dynamic call tree. Finally, by traversing the loop-free dynamic call tree in a depth-first order and keeping the visited nodes in a sequence, a loop-free execution trace is generated.

$$\frac{\ldots, Foo, F1, F1, F2, F1, F2, \ldots, F1, F2, \ldots}{\text{(a)}}$$

$$\frac{\ldots, Foo, F1, (F1, F2)^n, \ldots}{\text{(b)}}$$

$$\frac{\ldots, Foo, (F1)^2, (F2, F1)^{n-1}, F2, \ldots}{\text{(c)}}$$

Figure 5.6: (a) A string containing repetitions. (b) Representation of (a) in the form of one instance of string repetition. (c) Another possible representation of (a) in the form of one instance of string repetition.

In Procedure *Foo* a piece of code that produces a long trace with repetitions of "*F1, F2*" is shown. Figures 5.6(a) and 5.6(b) represent the parts of execution trace that is produced by Procedure Foo, and the result of applying Crochemore algorithm, respectively. In Figure 5.6(c) another representation for the string *F1,F2,F1,F2, ...,* *F1, F2* is shown. By following the above heuristic we would choose the loop free representation in Figure 5.6(b).

Procedure *Prune* describes the graph pruning algorithm. In this procedure, the string representation for the ID values of the input node's children is obtained by a call to the *getChildrenIDs* method. By applying the repetitive string finder algorithm Crochemore (calling the procedure *findRepetitions*), we identify the locations of the repeated items in this string, and consequently we remove them from the tree. By running this procedure on the root of the dynamic call tree, we will prune the whole dynamic call tree.

```
Procedure Prune(GraphNode node)
Input: GraphNode node
Result: pruned tree rooted at input GraphNode node
//Procedure findRepetitions is an implementation of the Crochemore algorithm, which
returns the locations of the repeated items in its input.;
begin
    $\mathcal{S} \longleftarrow node.getChildrenIDs()$;
    $indices \longleftarrow findRepetitions(\mathcal{S})$;
    foreach $i \in indices$ do
        delete corresponding child to $i$;
    end
    foreach $child \in node.getChildren()$ do
        $Prune(child)$;
    end
end
```

5.4 Execution Pattern Generation

In Section 4.1 we presented an implementation for the execution pattern miner $\Upsilon_n(R_{S_\phi})$ which takes a repository of pruned execution traces R_{S_ϕ} and generates the corresponding execution patterns P_{S_ϕ}. Each execution pattern reveals the common sequences of function invocations that exist within the different executions of a program that correspond to a set of task scenarios. In this context, we define a group of scenarios that all share a specific feature ϕ of the subject software system (namely a feature-specific scenario set S_ϕ) and execute them on the instrumented software system.

The group of loop-free execution traces, that are generated in the previous preprocessing steps, constitute the trace repository R_{S_ϕ}. We apply our sequential pattern miner Υ_n on R_{S_ϕ} where minimum support is set to $80\% * |R_{S_\phi}|$. In this setting the extracted execution patterns P_{S_ϕ} discover frequent sequences of function calls that exist in the majority of the execution traces (80% of them) and thus reveal the implementation of the feature(s) that exist in majority of task scenarios.

5.5 Execution Pattern Analysis

One characteristic of the aforementioned sequential pattern mining technique is that the extracted execution patterns P_{S_ϕ} reflect both the implementation point of the specific feature ϕ and the implementation points of the features that are necessary to set up every typical task scenario (examples of such features are initializing and termination of the software system). We employ a strategy to focus on execution patterns corresponding to specific features within each group of scenario sets. In order to do this, we first examine different features of the software system and store their corresponding execution patterns in a pattern repository. In a further analysis we identify those execution patterns that are specific to a single software feature within one scenario set, as well as those that are common among all sets of scenarios. In this section we first define two different types of functions that exist in execution patterns: feature-specific functions and omnipresent functions. Then two mechanisms for extracting each type of feature-specific/omnipresent functions are presented in Subsections 5.5.2 and 5.6, respectively.

5.5.1 Categories of Functions in Execution Patterns

An execution pattern is treated as a sequence of functions that implement common functionalities within a scenario set. In the following, the different kinds of patterns that exist in extracted execution patterns along with the corresponding extraction mechanisms are presented.

- **Feature-specific patterns**

  A feature-specific pattern corresponds to the core functions that implement a targeted feature ϕ of a feature-specific scenario set S_ϕ. Such a pattern exists in the majority of patterns of S_ϕ. In order to extract a feature-specific pattern, we should increase the level of $MinSupport$ of the generated execution patterns to a number that covers the majority of the scenarios in S_ϕ.

- **Omnipresent patterns**

  An omnipresent pattern is common to almost every task scenario of the software system (e.g., software initialization / termination operations, or mouse tracking). Such a pattern exists in every execution trace of every scenario-set S_ϕ. Therefore, it is extracted along with the feature-specific patterns mentioned above. In order to extract such a pattern, we should use a filtering mechanism (e.g. concept lattice in Section 4.2) to filter out the feature-specific patterns from this group of patterns.

Although each of the above categories may be required in a particular analysis task, the first category reveals the implementation of the feature that is targeted by the set of task scenarios and hence is considered as the more relevant type of dynamic analysis. Extraction of the feature-specific patterns and omnipresent patterns can be performed through two different strategies, as described below:

Strategy 1) given a feature-specific scenario set S_ϕ (sharing a specific feature ϕ) those sequences of functions that are executed during the majority of the scenarios are implementing the targeted feature(s) of the scenario set S_ϕ. In this strategy, we should increase the level of $MinSupport$ of the generated execution patterns to a number that covers the majority of the scenarios in the corresponding scenario set. In this context, the extracted execution patterns correspond to both feature-specific and omnipresent patterns.

Strategy 2) given a group of two or more feature-specific scenario sets, each with a different specific feature, the extracted execution patterns which are shared among the majority of the scenarios implement the common features of the software system.

In the rest of this chapter, we present two different filtering mechanisms to separate the omnipresent patterns from feature-specific ones. In Section 5.5.2, we discuss a filtering mechanism that is based on the second application of sequential pattern mining technique. In Section 5.6, we employ the visualization power of the concept lattice analysis to cluster feature-specific patterns corresponding to each particular feature.

5.5.2 Sequential Pattern Mining Approach

The generated execution patterns during the above-mentioned Strategy 1 are not pure, in the sense that they do not exclusively contain the functions related to the functionality of the specific feature of the scenario set. The omnipresent patterns mentioned above are also embedded within extracted patterns which must be identified and be separated. In doing this, we apply the sequential pattern mining algorithm for the *second time* on the result of the first execution pattern mining obtained from Strategy 1. The characteristics of the second pattern mining are described below:

Input characteristics:

- Each "execution pattern" in the result of the first pattern mining is considered as an "execution trace" for the second pattern mining, note each first generation pattern corresponds to highly repeated sequence of function calls in the original execution traces.
- The size of the input traces in the second pattern mining are much smaller than those in the first pattern mining.

Output characteristics:

In the result of the second pattern mining:

- functions that are participating in the patterns with small support are the feature-specific patterns.
- patterns with a large support correspond to the common sub-patterns in the first generated patterns that relate to omnipresent patterns mentioned above.

Omnipresent patterns may be embedded in feature-specific patterns and one has to identify and remove them. In doing so, we identify the locations of the second generation patterns within a first generation pattern, and record the number of appearances of functions in each second generation pattern within the first generation pattern. This number indicates the *support count* for each function. Note that this number is different from support of each execution pattern (i.e. size of support set of the execution pattern) since the functions in the overlap areas are counted. Thus, the support count of each function f reflects the level that f is shared among the different scenario sets. In this form, extracting functions that have fewer support count (e.g.,

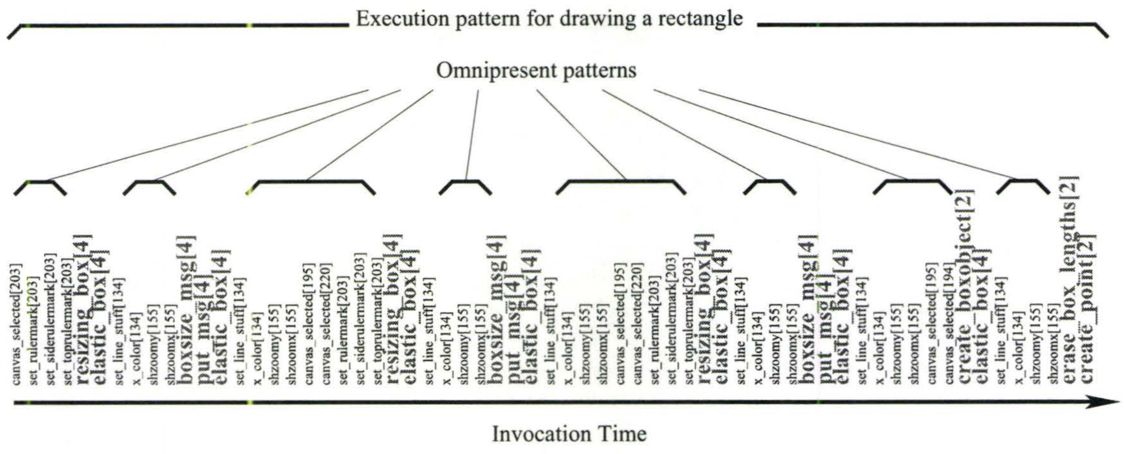


Figure 5.7: A first generation pattern extracted of drawing a rectangle in Xfig with the highlighted second generation patterns along with their support counts.

less than 5% of the number of the first generation patterns) signifies the extracted core functions of a specific feature within each original execution pattern. The rationale is as follows: these low-supported functions of execution patterns correspond to the singled-out targeted features of a scenario set that were extracted during the first pattern generation process. Similarly, the functions in second generation patterns with high support counts (e.g., more than 25% of the first generation patterns) signify the high frequency functions (utility operations) in a first generation pattern. Figure 5.7 depicts a part of a first generation pattern with the highlighted second generation patterns along with their support counts. The original execution pattern in this figure is extracted from a feature-specific scenario set that target the Xfig ability to draw a rectangle. The functions with bold fonts posses small support and perform significant role in specifying the boundary region for drawing a new rectangle on the screen.

An advantage of the method is that different groups of utility functions are extracted in different spots that enables the expert users to distinguish their functionality. Moreover, one can locate the extracted feature-specific/common patterns in the original execution traces and annotate the original trace with the corresponding extracted functionalities.

5.6 Concept Lattice Analysis Approach

We employ a strategy to spotlight on the execution patterns corresponding to specific features within a group of scenario sets. In this context, we use concept lattice analysis to cluster the group of functions in patterns that exclusively correspond to a shared feature of a scenario set; also to cluster the group of functions in patterns that are common to every scenario set.

5.6.1 Concept Lattice Construction

In Section 4.2, we define a formal context $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ as a triple which represents the relation $\mathcal{R}$ between objects $\mathcal{O}$ and their attribute values $\mathcal{A}$. In this chapter, we apply concept lattice analysis to represent the relation between features and functions such that $\Phi \equiv \mathcal{O}$ and $\mathcal{F} \equiv \mathcal{A}$. In our setting for concept lattice analysis, an object is a targeted feature $\phi \in \Phi$ of a feature-specific scenario set S_ϕ, and an attribute is a function f that participates in the execution patterns corresponding to S_ϕ. We focus our analysis on a subset of all features of the software system, and define Φ' to be the set of all analyzed features in this analysis. We construct the formal context $\mathcal{C}' = (\Phi', \mathcal{F}', \mathcal{R}')$ as belows:

- Let P_{S_ϕ} be the set of all execution patterns that are extracted with respect to feature-specific scenario set S_ϕ.

- Let F_ϕ be the set of all functions that exist in the extracted patterns P_{S_ϕ}.
- Construct the relation R_ϕ with respect to specific feature ϕ such that:

  $R_\phi = \{(\phi, f) | f \in F_\phi\}$.
- Create a formal context $\mathcal{C}' = (\Phi', \mathcal{F}', \mathcal{R}')$ such that:

  $\mathcal{R}' = \bigcup_{\phi \in \Phi'} R_\phi \quad and \quad \mathcal{F}' = \bigcup_{\phi \in \Phi'} F_\phi$.

5.6.2 Concept Lattice Analysis

Applying concept lattice analysis to the formal context described above will result in separation of omnipresent functions from functions that are specific to certain features. Since omnipresent functions are executed through almost every task scenario of the software system, these functions exist in the intent of almost every concept of the lattice and consequently appear in upper region of the lattice. On the other hand, functions that are specific to certain features of the software (feature-specific functions) are located in lower region of the lattice.

Moreover, a concept whose extent consists of a single object (feature ϕ) collects functions that exclusively implement feature ϕ. In other words, these functions represent the logical module that implement feature ϕ in the software system. In the following, we define the group of concepts that are relevant to feature-specific function clusters.

- *Feature-specific concept* c_ϕ is a concept whose extent consists of a single feature ϕ.
- We define F'_ϕ to be the set of functions that label c_ϕ on the concept lattice,

thus:

$$F'_\phi = \{f \in F_\phi | \mu(f) = c_\phi\}$$

where $\mu(f)$ is the function that returns the most general concept that has f in its intent (see Section 4.2).

In the generated lattice all the common functions are clustered in the upper region of the lattice, however disables the analysis to distinguish different group of common functions that are associated with different functionalities. As opposed to the second pattern generation mechanism which requires the user interaction to decide the functionality of each group of extracted core functions, concept lattice clusters the feature-specific functions within feature-specific concepts. Consequently, functionality of the extracted functions can be easily identified using the specific feature of the corresponding feature-specific concept.

Chapter 6

Structural Evaluation of Software System

Software systems are continuously evolving throughout their lifetime from early development to their maintenance and retirement. During the maintenance phase the software system is still changing through activities such as bug-fixing, migration to new platforms, and adding new features which were not planned from the beginning. Therefore, even a nicely designed and accurately implemented software system will probably incur several changes to its functionality and consequently to its structural design. This common scenario is the main cause of structural damage, high maintenance cost, and eventually retirement of a legacy system. To help this situation, the task of the software maintainers is to measure the impact of the newly added features on the structure of the software system. In this context, the maintainers can make sure that the newly added features will not damage the original structure of the software system.

One approach to address this problem is to assess the structural merit of the software system based on the degree of functional scattering of software features among the structural modules. In this context, the functionality of the system is represented as a set of features that are implemented within the software modules and are manifested as constituents of different scenarios to be run on the software system. In addition, the functional cohesion of each system module can also be investigated as a means to monitor the healthiness of the software system.

In this chapter, we provide two metrics to assess the structural merit of the software system: *feature functional scattering* and *structural cohesion*. The proposed feature functional scattering metric examines the distribution of a set of functions that implement a specific feature over the structural units (i.e., files) of the system. Hence, it represents the degree of scattering of the implementation of software features among the structural modules. On the other hand, the structural cohesion assessment directly represents the cohesion of module(s) implementing a specific feature based on the functional relativeness of the functions that reside in each structural unit (module); this measure of cohesion is much closer to the original definition of cohesion ("relative functional strength of a module" [22]) than using static structural techniques such as inter-/intra-edge connectivity of the components.

6.1 Metrics Computation

In order to measure the feature functional scattering of feature ϕ, we assess the degree of distribution of collected functions of logical module F'_ϕ over the structure of the system. Moreover, we compute the functional relativeness of the functions that reside in each module in order to evaluate the module structural cohesion. In doing so, the set of functions that implement a certain feature ϕ are extracted from the above discussed concept lattice analysis (i.e. F'_ϕ). Then, the source files in which these functions are defined are identified and the ratio of the number of functions that are used from each file to the number of functions that are defined in that file is calculated. This ratio is a measure of structural cohesion of the system files that contribute to implementing the feature under study.

6.1.1 Formal Definitions

In this subsection, we provide exact definitions for the aforementioned structural merit evaluation metrics, where $SC_\phi(m)$ denotes structural cohesion of module m with respect to logical module F'_ϕ and $FS(\phi)$ denotes functional scattering of feature ϕ.

- Let $M_\phi = \{m_1, m_2, \ldots, m_k\}$ be the set of modules where all the functions in F'_ϕ are defined in elements of M_ϕ.
- Let F_m denotes the set of functions that are defined in module m.
- Structural cohesion of module m with respect to logical module F'_ϕ, namely $SC_\phi(m)$, is defined as:
$$SC_\phi(m) = \frac{|F_m \cap F'_\phi|}{|F_m|}$$
- Functional scattering of feature ϕ, namely $FS(\phi)$, is defined based on the distribution of functions in F'_ϕ over modules in M as:
$$FS(\phi) = 1 - \frac{\sum_{m \in M_\phi} SC_\phi(m)}{|M_\phi|}$$

6.2 Discussion

A software system with high structural cohesion $SC_\phi(m)$ for its individual modules and low functional scattering $FS(\phi)$ among its structure represents a modular system that requires less maintenance efforts. However, a high degree of functional scattering corresponding to feature ϕ directly signifies a high structural impact that is caused by that feature. Hence the system requires more maintenance efforts to tackle with the consequences of propagated change to other software modules.

Note that feature functional scattering and structural cohesion metrics are not standalone metrics and must be considered as a whole. A low degree of functional scattering corresponding to feature ϕ solely do not imply a good modular structure whereas ϕ could be defined in more than one highly cohesive module.

Chapter 7

Experiments

In this chapter, we apply the proposed dynamic analysis technique on two medium-size open source systems that are discussed in the following sections. The developed dynamic analysis tool is an Eclipse plug-in [3] and has been developed as an extension to the Alborz reverse engineering toolkit [26] to enhance the scope of Alborz to cover both static and dynamic analysis of a software system. In Section 7.1, we discuss the results of our analysis using the Xfig [1] drawing tool. In Section 7.2 the results of our analysis using Pine [2] email client are presented.

7.0.1 Platform

The hardware platform for the experiments consists of a Pentium II with 440 MHZ CPU and 512M bytes memory which runs a Red Hat Linux 7.3. This machine is used for instrumenting the subject systems, executing the feature-specific scenarios, and capturing the raw function entry/exit pairs. The actual analysis process for extracting the execution patterns and performing pattern analysis is done on a Windows XP professional edition which runs on a laptop with a 1.5GHZ Centrino processor, 512M bytes memory, and 1G bytes virtual memory.

7.1 Dynamic Analysis of Xfig

Xfig 3.2.3d [1] is an open source, medium-size (80 KLOC), menu driven, C language drawing tool under X Window system. Xfig has the ability to interactively draw and manipulate graphical objects (circle, ellipse, line, spline, rectangle, and polygon) through operations such as copy, move, delete, edit, scale, and rotate. In the following we discuss the steps of applying the proposed dynamic analysis technique on the Xfig drawing system.

7.1.1 Feature-Specific Scenario Generation

In order to extract the core functions that implement a specific feature (e.g., *flip* in Table 7.1) we define a group of feature-specific scenarios to target this feature and execute on the instrumented Xfig system to obtain the corresponding function entry/exit pairs. Figure 7.1 depicts the adopted strategy to single out a targeted feature by means of a set of task scenarios. In this setting, a group of seven scenarios have been selected that all begin from the start up operation and finish in the terminate operation. Each scenario has a distinct path within the Drawing component, but shares the same path (i.e., flip operation) within the Editing component. The group of task scenarios shown in Figure 7.1 form a feature specific scenario set, where the *flip* operation is the specific feature. We apply the above strategy to generate feature-specific scenario sets that each target one feature of the Table 7.1.

7.1.2 Execution Pattern Extraction

For each feature-specific scenario set S_ϕ, we execute the scenarios of S_ϕ on the instrumented Xfig system and obtain the corresponding function entry/exit pairs. After pruning the extracted entry/exit pairs from loop-based function calls (Section 5.3.1)

Feature Family	Specific Xfig Feature	Number of Scenarios	Average Trace Size	Average Pruned Trace Size	Number of Extracted Patterns	Average Pattern Size
Draw Ellipse	Circle-Diameter	10	7234	2600	46	33
	Circle-Radius	10	8143	2463	48	32
	Ellipse-Diameter	10	6405	2536	41	37
	Ellipse-Radius	10	7351	2549	39	35
Copy	Move Objects	4	11887	3166	31	53
	Copy Objects	4	11460	3269	37	50
Draw Spline	Closed Interpolated	10	18635	4434	58	63
	Interpolated	10	15469	4038	66	49
	Approximated	10	15057	5362	61	47
Scale	Center Scale	4	8088	1541	30	47
Flip	Flip up-Right	4	7296	1378	29	46
Rotate	Rotate Clockwise	4	6974	1544	28	44
Delete	Delete Objects	4	6580	1181	19	56

Table 7.1: The result of execution trace extraction and execution pattern mining for a collection of 7 Xfig feature families and their specific features.

we apply the execution pattern mining process to obtain the patterns of function call sequences. Table 7.1 presents the statistical information for the experimented features of the Xfig system.

7.1.3 Concept Lattice Analysis

In this analysis, we supply the resulting execution patterns of the Xfig experiments to a concept lattice generation tool (*concept explorer* [4]) in order to view the distribution of the feature functions on the lattice. As it was discussed in Section 5.5 the feature-specific concepts (i.e., a concept whose extent consists of a single feature) remain in the lower region of the lattice, and collect the functions that exclusively implement specific features. Similarly, concepts with omnipresent functions (i.e., a concept which is labeled with functions that are shared among a majority of concepts) appear in

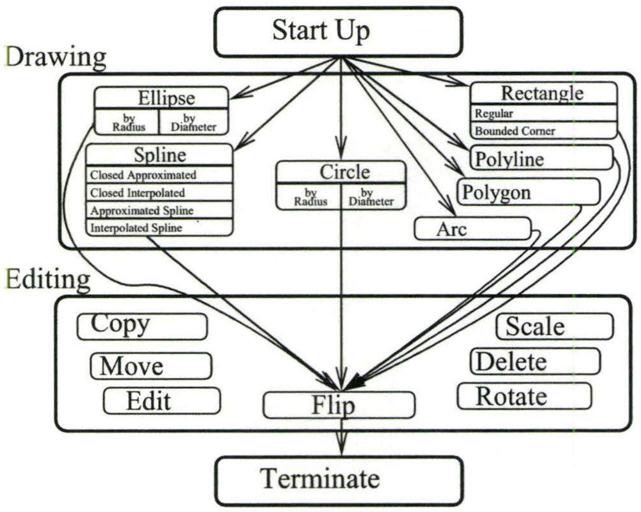


Figure 7.1: A Feature-specific scenario set for Xfig drawing tool. The group of scenarios apply the *Flip* operation on different graphical objects.

the upper region of the lattice. Viewing the distribution of the concepts and their functions throughout the concept lattice allows to get insight into the structure of the feature-specific concepts and their functions.

Consequently, it allows us to collect the group of functions that correspond to different feature-families. In Figure 7.2 three dashed circles at the bottom illustrate the group of concepts and their functions that implement the core functionality of the feature families of *ellipse, copy*, and *spline.*

7.1.4 Structural Evaluation

Finally, based on inspecting the source files of Xfig, we measure the structural cohesion of corresponding source files, as well as the feature functionality scattering of the features under study. The results of this evaluation for three feature families *Draw Ellipse*, *Copy*, and *Draw Spline* are presented in Table 7.2.

For the three mentioned feature families we inspect the Xfig source files that define

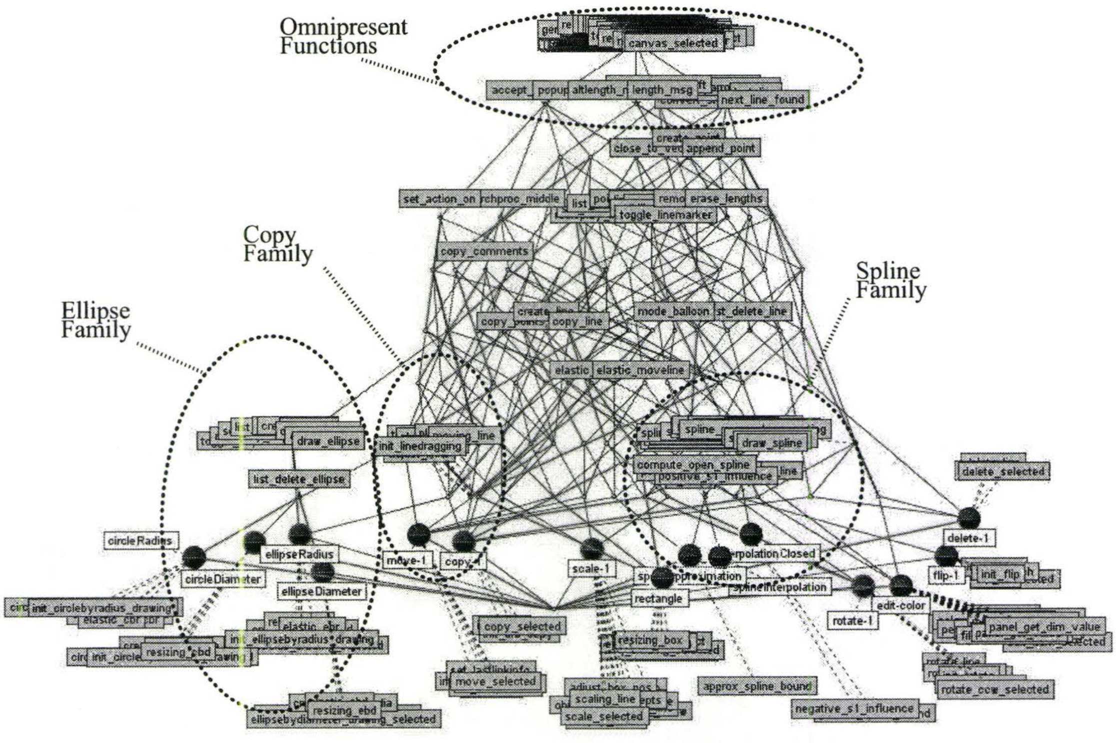


Figure 7.2: Concept lattice representation of the extracted features and their corresponding functions for the Xfig drawing tool. The group of concepts corresponding to three feature families and the omnipresent functions are shown by dashed ovals.

the functions that implement the corresponding logical module of that feature family. The results of measuring the structural cohesion $SC_{\phi}(m)$ of these files are presented in Table 7.2. These results indicate that file *d_ellipse* has high cohesion with respect to logical module of feature family *Ellipse*; files *e_copy*, and *e_move* are also highly cohesive with respect to feature family *Copy*; and finally, file *d_spline* is cohesive with respect to feature family *Spline*. However, study of the feature functional scattering measures allows us to better interpret the characteristics of these logical modules. For example, in the case of *Ellipse* a portion of the logical module is located in a large structural module *u_elastic* which results in a high functional scattering measure. Whereas, in the case of *Copy* feature family, the logical module almost covers two

Feature ϕ	**Contributed File** (m)	$\lvert F_m \rvert$	$\lvert F_m \cap F'_\phi \rvert$	**Structural Cohesion** $SC_\phi(m)$	**Functional Scattering** $FS(\phi)$
Ellipse	d_ellipse.c	16	12	75%	57%
	u_elastic.c	67	8	12%	
Copy	e_copy.c	5	3	60%	32%
	e_move.c	4	3	75%	
Spline	d_line.c	9	2	22%	66%
	d_spline.c	6	5	83%	
	u_bound.c	19	2	11%	
	u_draw.c	75	14	19%	

Table 7.2: Structural cohesion and feature functional scattering measures for three different feature families of the Xfig.

structural modules *e_copy* and *e_move* which indicates low scattering.

In the case of *Spline*, the logical module is almost equally scattered among four structural modules each covering a small portion of the structural modules and hence indicating high functional scattering. The results in Table 7.2 are promising in the sense that they reflect meaningful measures with respect to the sizes of logical and structural modules shown. Regarding the results of our structural evaluations, we can predict high maintenance activities regarding any change to the feature families *Ellipse* and *Spline*. Similarly, changes to the *Copy* feature family would not propagate throughout the system which indicates less maintenance activities.

7.1.5 Characteristics

In the followings, we discuss the important properties of the proposed pattern based dynamic analysis technique using the Xfig case study.

- *Mapping logical modules onto structural modules*

  Table 7.3 demonstrates the results of experimentation with Xfig tool to reveal the core functions for three Xfig features. We focus on drawing a figure in the ellipse family includes circle, ellipse and such that each figure can be drawn in two different ways, i.e., by-radius and by-diameter. Furthermore, we expand our experiments on an editing operation of the Xfig tool (i.e., copy graphical objects) as well as another family of graphical objects (i.e., spline). The extracted logical modules are shown in Table 7.3 and according to the Xfig naming convention it is clear that the logical modules truly reflect the core functions of the feature families.

- *Focusing on the important sub-traces*

  Table 7.1 represents the attributes of a group of feature-specific scenario sets that we use in the analysis process. This table illustrates a major characteristic of the proposed dynamic analysis with regard to reducing the scope of the analysis from huge sizes of the execution traces (Average Trace Size) to the manageable sizes of the execution patterns (Average Pattern Size).

- *Separating common patterns from feature-specific patterns*

  Figure 7.2 illustrates the mapping of extracted Xfig's feature related functions on the concept lattice. In this lattice, the upper nodes collect omnipresent functions of Xfig corresponding to common patterns, including: software initialization and termination, mouse pointer handling, canvas view updating, and side ruler management. In addition, the specific functions that exclusively implement a feature are located in the lower region of the concept lattice through feature-specific concepts. Table 7.3 represents the core functions that implement the certain family features of Xfig (i.e., Copy Object, Draw Ellipse, Draw Spline).

Feature Family	**Extracted Core Functions representing logical module F_{Φ_ϕ}**
Ellipse	init_circlebyradius_drawing, elastic_cbr, resizing_cbr, create_circlebyrad, circlebyradius_drawing_selected, init_circlebydiameter_drawing, elastic_cbd, resizing_cbd, create_circlebydia, circlebydiameter_drawing_selected, init_ellipsebydiameter_drawing, elastic_ebd, resizing_ebd, create_ellipsebydia, ellipsebydiameter_drawing_selected, init_ellipsebyradius_drawing, elastic_ebr, resizing_ebr create_ellipsebyrad, ellipsebyradius_drawing_selected, add_ellipse, pw_curve, create_ellipse, center_marker, draw_ellipse, redisplay_ellipse, ellipse_bound, list_add_ellipse, set_latestellipse, toggle_ellipsemarker, list_delete_ellipse
Copy	copy_selected, init_copy, init_arb_copy, set_lastlinkinfo, init_arb_move, init_move, move_selected, set_lastposition, set_newposition, moving_line, init_linedragging, adjust_pos, place_line, translate_line, adjust_links, place_line_x
Spline	spline_drawing_selected, init_spline_drawing, get_intermediatepoint, elastic_line, unconstrained_line, toggle_splinemarker, h_blend, g_blend, draw_spline, spline_bound, step_computing, point_computing, spline_segment_computing, point_adding, create_splineobject, redisplay_spline, negative_s2_influence, next_spline_found, valid-spline_in_mask, init_trace_drawing, init_spline_drawing2, last_spline list_add_spline, create_spline, make_sfactors, create_sfactor, add_spline, set_latestspline, positive_s2_influence, positive_s1_influence, compute_open_spline, f_blend, negative_s1_influence, general_spline_bound, aprox_spline_bound, compute_closed_spline

Table 7.3: Results of dynamic analysis on Xfig drawing tool. The core functions (right column) correspond to the specific Xfig features (left column).

7.2 Dynamic Analysis of Pine

Pine 4.4.0 [2] is an open source, medium-size (207 KLOC), C language email client. Pine is a tool for reading, sending, and managing electronic messages. Feature functionalities of Pine can be categorized as belows.

- Online help specific to each screen and context.
- Message index showing a message summary which includes the status, sender, size, date and subject of messages.
- Commands to view and process messages: Forward, Reply, Save, Export, Print,

Specific Feature	Number of Scenarios	Average Trace Size	Average Pruned Trace Size	Number of Extracted Patterns	Average Pattern Size
Compose	8	90081	24636	95	172
Folder List	4	48335	11205	25	491
Message Index	5	67741	19529	44	345
Address Book	3	59221	16024	71	212

Table 7.4: The result of execution trace extraction and execution pattern mining for a collection of 4 different Pine features.

Delete, capture address, and search.

- Message composer with easy-to-use editor and spelling checker.
- Address book for saving long complex addresses and personal distribution lists under a nickname.
- Message attachments via the Multipurpose Internet Mail Extensions (MIME) specification. MIME allows sending/receiving non-text objects, such as binary files, spreadsheets, graphics, and sound.
- Folder management commands for creating, deleting, listing, or renaming message folders. Folders may be local or on remote hosts.

In our case studies, we examine four different features of the Pine for composing emails, managing the folder lists, address book, and message index. In order to extract the core functions implementing a specific feature, we define a group of scenarios to target that feature and consequently extract the corresponding group of functions through execution pattern extraction process (see Table 7.4). By repeating this process and targeting other features of the system with proper sets of scenarios, we would incrementally explore the system's overall functionality. By spreading the extracted execution patterns over a concept lattice (see Figure 7.3), we separate the omnipresent functions from specific functions that implement experimented features (see Table 7.5). Finally, based on inspecting the source code of the Pine, we measure the distribution of functions implementing each examined feature over the structural units (see Table 7.6).

Feature Family	**Extracted Core Functions representing logical module**
Address book	ab_resize addr_book addr_book_screen adrbk_check_all_validity_now adrbk_check_and_fix adrbk_check_local_validity adrbk_check_validity adrbk_num_from_lineno adrbk_write ae calculate_field_widths cur_addr_book cur_is_open display_book dlc_next dlc_prev dlc_siblings draw_cancel_keymenu end_adrbks entry_is_addkey entry_is_askserver entry_is_clickable entry_is_listent erase_checks erase_selections file_attrib_copy first_line flush_dlc_from_cache get_display_line get_top_dl_of_adrbk hashtable_size in_dir init_adrhash_array intr_handling_on intr_proc is_empty menu_clear_cmd_binding menu_init_binding paint_line rd_check_remvalid rename_file skip_to_next_nickname tempfile_in_same_dir temp_nam was_nonexistent_tmp_name write_hash_header write_hash_table write_hash_trailer write_single_abook_entry write_single_entryref
Folder list	compare_names context_screen endbold folder_lister folder_lister_km_manager folder_list_handle folder_list_text folder_list_write folder_list_write_prefix folder_processor folder_screen folder_select_preserve folder_select_restore free_handle free_handle_locations handle_on_page new_handle q_status_message2 redraw_scroll_text refresh_folder_list reset_context_folders scroll_handle_obscured scroll_handle_set_loc selected_folders
Message index	body_parameter body_type_names clear_cur_embedded_color color_a_quote decode_text describe_mime format_blip_seen format_message format_mime_size format_size_guess gf_control_filter gf_escape_filter gf_line_test gf_line_test_free_ins gf_line_test_opt mail_view_screen next_attachment percentage pine_header_standard rfc1738_scan rfc2231_get_param rfc2369_editorial so_nputs strsquish type_desc url_hilite url_hilite_hdr view_writec view_writec_destroy view_writec_init view_writec_killbuf web_host_scan zero_atmts

Table 7.5: Part of the results of dynamic analysis on Pine email client. The core functions (right column) correspond to the specific Pine features (left column).

For each feature in Table 7.4 we inspect the Pine source files that define the functions that implement the corresponding logical module. The results of measuring the structural cohesion $SC_\phi(m)$ of these files are presented in Table 7.6. These results

Feature ϕ	**Contributed File** (m)	$\lvert F_m \rvert$	$\lvert F_m \cap F'_\phi \rvert$	**Structural Cohesion** $SC_\phi(m)$	**Functional Scattering** $FS(\phi)$
Compose	context.c	13	2	16%	
	bldaddr.c	78	9	12%	
	send.c	99	57	56%	
	reply.c	65	12	19%	74%
Folder List	folder.c	121	15	12%	88%
Address Book	adrbklib.c	88	12	14%	
	addrbook.c	75	20	27%	80%
Message Index	pine/mailview.c	126	21	17%	83%

Table 7.6: Structural cohesion and feature functional scattering measures for four different features the Pine email client.

indicate high degree of scattering and low coupling among the examined feature families of Pine. Files *context, bldaddr*, and *reply* has low cohesion with respect to logical module of feature *Compose*; file *send* shows high cohesion with respect to feature *Compose*. However, study of the feature functional scattering measures allows us to better interpret the characteristics of these logical modules. For example, in the case of *Compose* a portion of its logical module is located in a large structural module *send* which results in a high functional scattering measure.

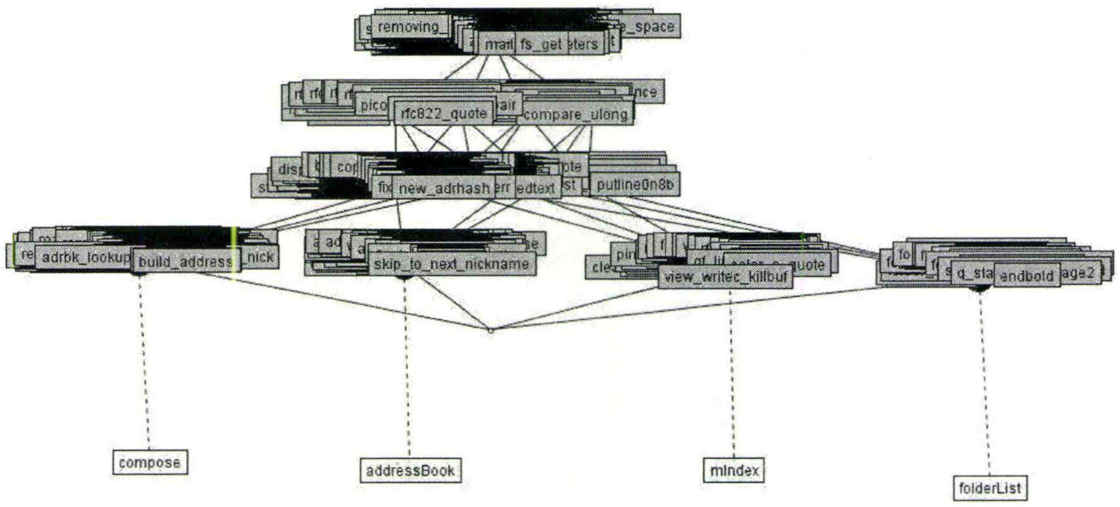


Figure 7.3: Concept lattice representation of the extracted features and their corresponding functions for the Pine email client.

Chapter 8

Conclusion and Future Work

In this thesis, we proposed a novel approach to dynamic analysis and structural assessment of a software system that takes advantage of frequent patterns of execution traces that exist within the executions of a set of carefully designed task scenarios. The proposed approach benefits from the discovery nature of data mining techniques and concept lattice analysis to extract both feature specific and common functions that implement important features of a software system. The resulting execution patterns provide discovery of valuable information out of noisy execution traces. The proposed approach is centered around a set of task scenarios that share a specific system feature and introduces a means for measuring the impact of individual features on the structure of the software system. The whole process consists of several steps such as: software instrumentation; feature-specific scenario set selection; loop-based execution trace elimination; execution pattern extraction; and finally structural assessment of the software system. The proposed technique has been applied on two medium size interactive software systems with very promising results in extracting both feature-specific and common functions. Moreover, the level of "structural cohesion" and "feature functional scattering" are measured that provide a way for assessing the structure of the experimented tools.

8.1 Discussion

In this section, we discuss the characteristics of the proposed sequential pattern analysis. With regard to our definition for an execution pattern as a continuous sequence of function calls, we extract core functions that implement specific functionalities of the system. By extending the definition of the execution pattern to include noncontinuous function invocations, we can extract function patterns that implement more general functionalities; however such an expansion may result in extracting meaningless execution patterns (by joining unrelated parts of the execution trace to form a new pattern) and generating an overwhelming number of patterns. The general algorithms for sequential pattern mining in the data mining literature would allow extracting patterns that have functions interleaved with the extracted patterns. The study of trade-off between discovering execution patterns that implement more general functionality and dealing with an overwhelming number of extracted patterns would be a more challenging problem that is listed in our future work tasks.

Moreover, we can employ other pattern mining techniques such as tree-pattern mining, where the pattern miner looks for identifying patterns that exist among dynamic call trees as opposed to our technique that identifies patterns among execution traces.

8.2 Future Work

Currently, we apply our technique to the problem of feature identification, however the application of execution patterns in software architecture recovery by augmenting the current static analysis technique must also be considered. This will make a hybrid technique that enhances the power of static analysis techniques such as clustering, pattern matching, and concept lattice analysis with dynamic analysis information of the software under investigation. The result of the proposed dynamic analysis technique can be used to incorporate semantics to the existing static analysis techniques. The future tasks include the investigation of noncontinuous execution patterns as well as proposing effective pruning methods at the execution trace generation to allow analysis of very large traces over 100K functions. (e.g. Apache, MySql).

Appendix A

Tool Documentation

We design the Dynamic Alborz toolkit as a data centralized and user interface driven architecture. Six components are designed to collaborate with each other and fulfill the functionalities of the system. These components are User Interface (UI); Datastore; Preprocessing; Pattern Mining; Post-processing; and Environment. Figure A.1 is a standard UML component diagram which describes the detail of each component and relationships among them. The environment component in the diagram does not represent a concrete component in the system, but some external tools used by the system, e.g. instrumentation tools used for extracting execution traces and concept lattice tool used for lattice visualization.

A.1 Architectural Design

In the following sections, we elaborate the functionalities and interface of each component.

User Interface Component

UI closely collaborates and controls other components within the Dynamic Alborz

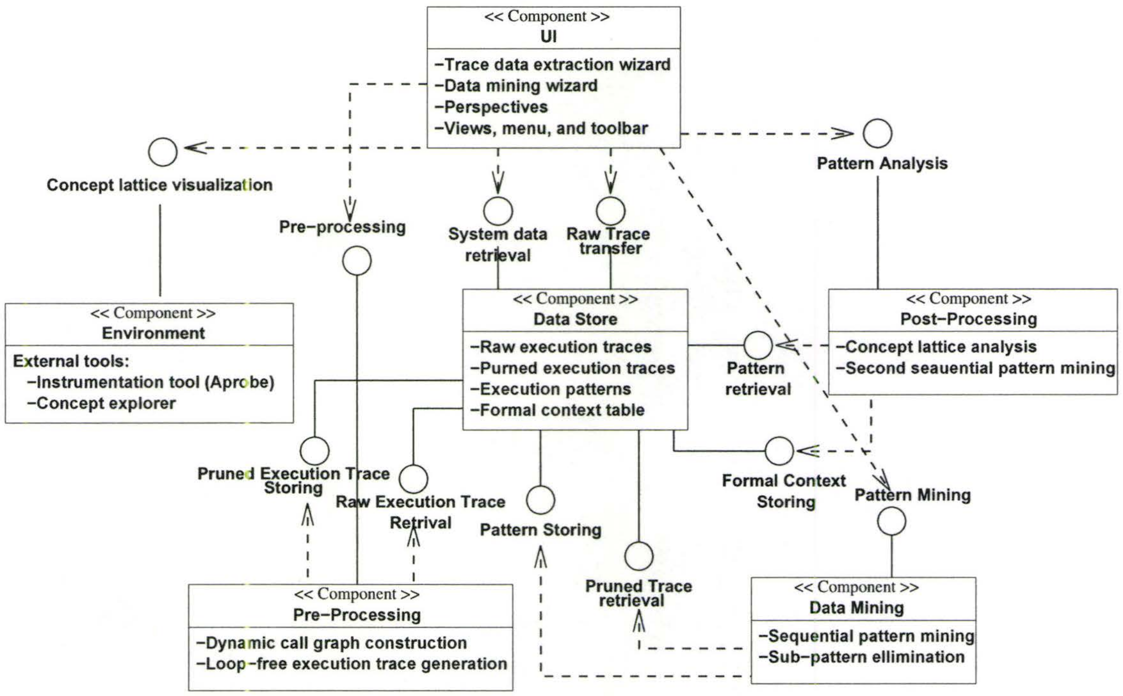


Figure A.1: Component digram of the Dynamic Alborz plugin in the Eclipse environment.

plugin and interacts with the user throughout the analysis phases. Almost all the events and requests in the system are emitted from this component. UI consists of the following parts:

- Dynamic Alborz Run Wizard: This wizard helps the user to perform the main task of the Dynamic Alborz. This wizard consists of the following parts:
  - *Trace data extraction page*: which controls the operations of the *preprocessing component* by guiding the operations within the toolkit. It allows the user to: i) extract execution traces for dynamic analysis; ii) preprocess execution traces and eliminate loop-based redundancies; vi) store system data in the local Datastore to be used for the further analysis phases.

- *Data mining page*: that assists the user throughout the steps for the sequential pattern mining operation, such as: minimum support selection.
- *Pattern Analysis page*: that controls the pattern analysis process and allows the user to select system features that should be included in a specific analysis session.

- *Config Wizard*: This wizard helps the user to set up the environmental variables of the Dynamic Alborz, such as: system work path, and path to the Concept lattice analysis tool.

- *Perspectives, views, menus and tool-bars*: a series of standard Eclipse user-interface elements used for integrating Dynamic Alborz with the Eclipse platform, such as: system data navigator, feature view, pattern analysis result view.

Datastore Component

Datastore is the center of the Dynamic Alborz structure and allows the system components to communicate with each other through this component's interfaces. Files are used for storing: raw execution traces; pruned execution traces; raw execution patterns; formal contexts; and results of the entire dynamic analysis. This component acts as an intermediate object which connects other components in the system. All other components which need to store data or retrieve data communicate with this component through its interfaces. Internally, the datastore component stores everything in a directory structure which uses the local data store as its underlaying media.

Post-processing Component

Encapsulates all the functionalities that are required in the pattern analysis phase including:

- execution pattern translation, that provides statistical information about the extracted execution patterns such as number of extracted patterns and average pattern length, it also locates each execution pattern in its corresponding execution traces
- formal context generation, that parses the generated execution patterns and generates formal context tables
- second pattern generation, that parses the generated execution patterns and applies the second sequential pattern mining.

Preprocessing Component

The preprocessing component encapsulates the functionalities and algorithms required for dynamic call graph construction and loop-free execution trace construction. An implementation of the Crochemore string processing algorithm is used for finding the loop-based execution traces.

Data mining Component

Provides an implementation of the sequential pattern mining algorithm that parses the pruned execution traces and extracts sequential patterns among them. It also prunes the generated execution patterns and eliminates sub-patterns.

A.2 Design Pattern

In this section, we discuss the approach that we used for enhancing the structure of the prototype Dynamic Alborz system and hence obtaining the extensibility. Figure A.2 describes how Observer pattern is used when we integrate Dynamic Alborz toolkit with Eclipse platform. The interface IProgressMonitor is the observer and IRunableWithProgress is the subject to be observed.

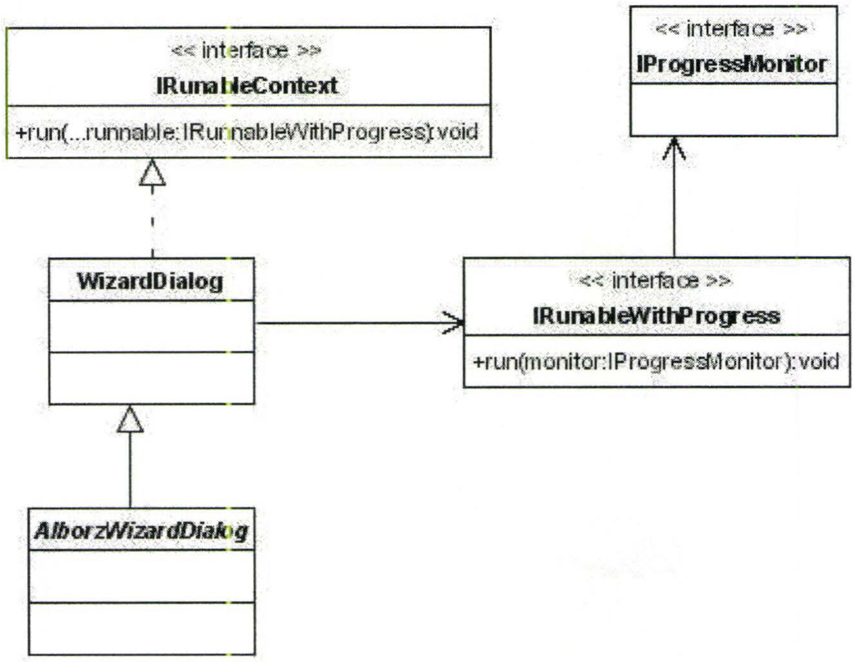


Figure A.2: Observer design pattern used in the Dynamic Alborz plugin.

A.3 User Interface Design

Dynamic Alborz is designed as an Eclipse plugin that makes its usage and deployment an easy task. The Dynamic Alborz plugin constructs the following user interface parts in order to integrate with the Eclipse environment:

1. Dynamic Alborz Menu
2. Dynamic Alborz Perspective which includes System Navigator view, Feature view, Pattern Analysis view, and Progress view
3. Config Wizard that provides a wizard-based user interface that eases the configuration of the system
4. Dynamic Alborz Run Wizards that performs the analysis process.

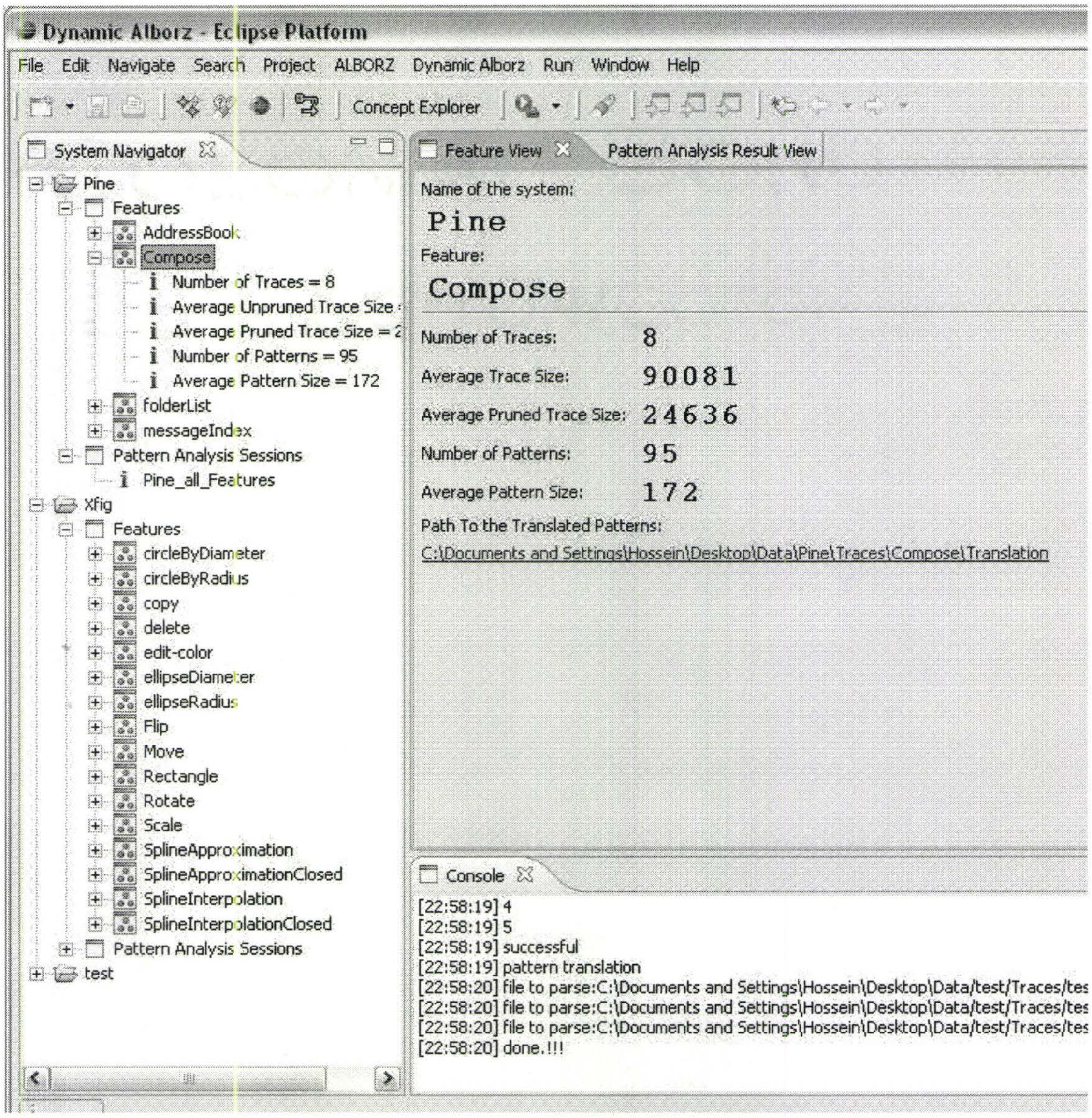
Figure A.3 provides a comprehensive overview of the Dynamic Alborz plugin inside the Eclipse environment. In the following we explain the Dynamic Alborz Run wizard and Config wizard, respectively.

Config Wizard

The Dynamic Alborz Config wizard is designed to ease the configuration of the dynamic analysis. Using this wizard the user can set the path for the current working directory of the system as well as the directory path of the Concept Explorer toolkit. Figure A.4 depicts this wizard inside the Eclipse environment.

Dynamic Alborz Run Wizard

The Dynamic Alborz Run wizard provides a wizard-based user interface that utilizes the dynamic analysis. Using this wizard the user can create a new subject system for analysis or select an already analyzed system (see Figure A.5). It also provides a history of the analyzed features of the selected system (see Figure A.6) that gives the user the choice between adding a new feature to the analyzed features of the selected system or starting the pattern analysis process for the selected system. In Figure A.7 the wizard page that provides the interface for importing a new feature for analysis in the system is shown. In this page the user provides the name of the new feature, a description of the feature and its corresponding feature-specific scenarios, and the path to the pruned execution traces for this new feature. After providing the required information for the new feature the system prunes the execution traces and stores them in its internal data structures. In the next step the user can select the minimum support of the data mining operation (see Figure A.8). The results of the pattern mining operation is shown in the "Statistical Results" page (see Figure A.9). In this page the user again has the choice between starting the pattern analysis process or adding a new feature for analysis to the system. In the "Pattern Analysis" page the user has the choice to choose among the analyzed features of the current system and specify a specific analysis session consisting of the selected features. Currently, the system implements the Concept Lattice Analysis however in the near future the required components for the Second Pattern Mining would be added (see Figure A.10).



Name of the system: Pine

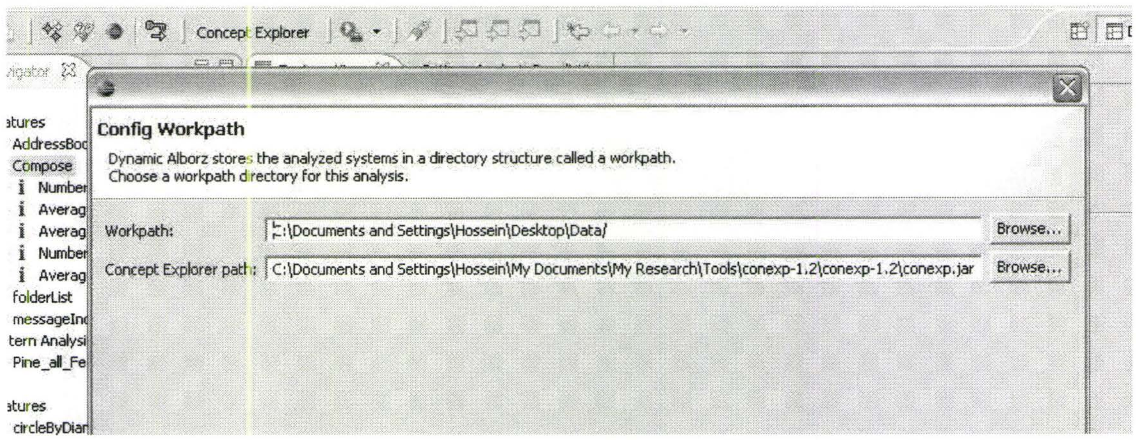
Feature: Compose

Number of Traces:	8
Average Trace Size:	90081
Average Pruned Trace Size:	24636
Number of Patterns:	95
Average Pattern Size:	172

Path To the Translated Patterns:
C:\Documents and Settings\Hossein\Desktop\Data\Pine\Traces\Compose\Translation

```
[22:58:19] 4
[22:58:19] 5
[22:58:19] successful
[22:58:19] pattern translation
[22:58:20] file to parse:C:\Documents and Settings\Hossein\Desktop\Data/test/Traces/tes
[22:58:20] file to parse:C:\Documents and Settings\Hossein\Desktop\Data/test/Traces/tes
[22:58:20] file to parse:C:\Documents and Settings\Hossein\Desktop\Data/test/Traces/tes
[22:58:20] done.!!!
```

Figure A.3: The Eclipse workbench with the Dynamic Alborz plugin installed.



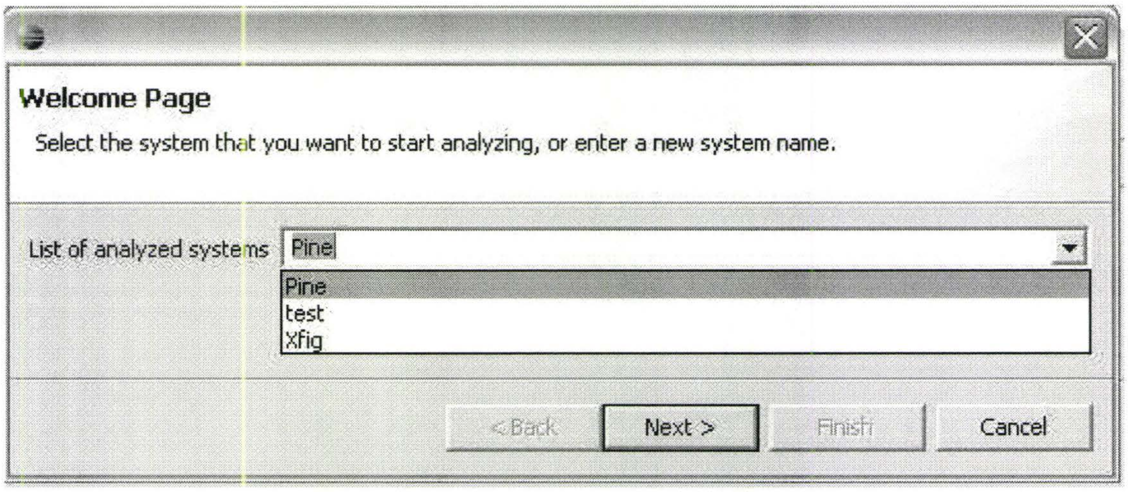
Config Workpath

Dynamic Alborz stores the analyzed systems in a directory structure called a workpath.
Choose a workpath directory for this analysis.

Workpath: C:\Documents and Settings\Hossein\Desktop\Data/

Concept Explorer path: C:\Documents and Settings\Hossein\My Documents\My Research\Tools\conexp-1.2\conexp-1.2\conexp.jar

Figure A.4: Config Wizard page of the Dynamic Alborz plugin.

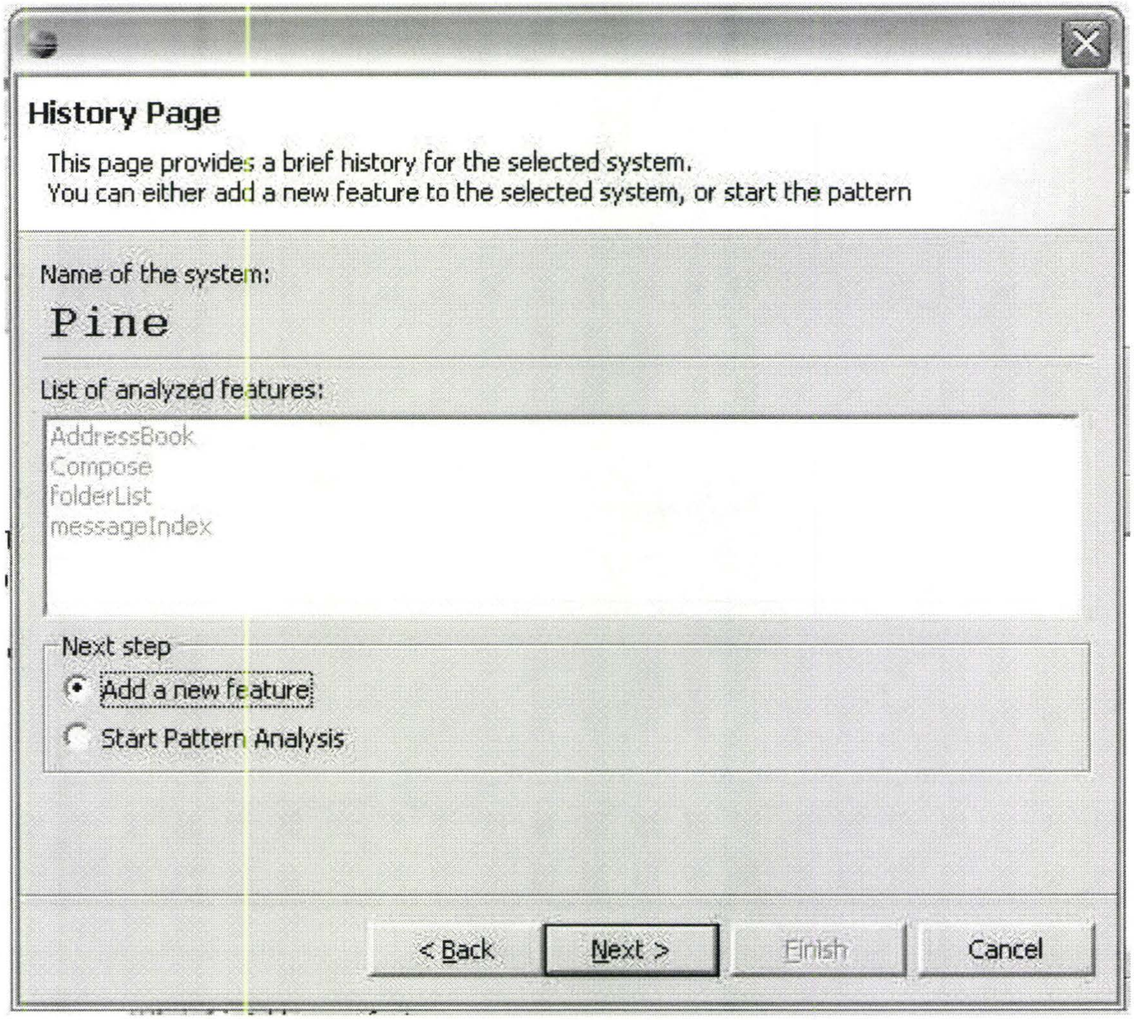


Welcome Page

Select the system that you want to start analyzing, or enter a new system name.

List of analyzed systems: Pine

Figure A.5: Welcome page, the first page in the Dynamic Alborz Run wizard.



History Page

This page provides a brief history for the selected system.
You can either add a new feature to the selected system, or start the pattern

Name of the system:

Pine

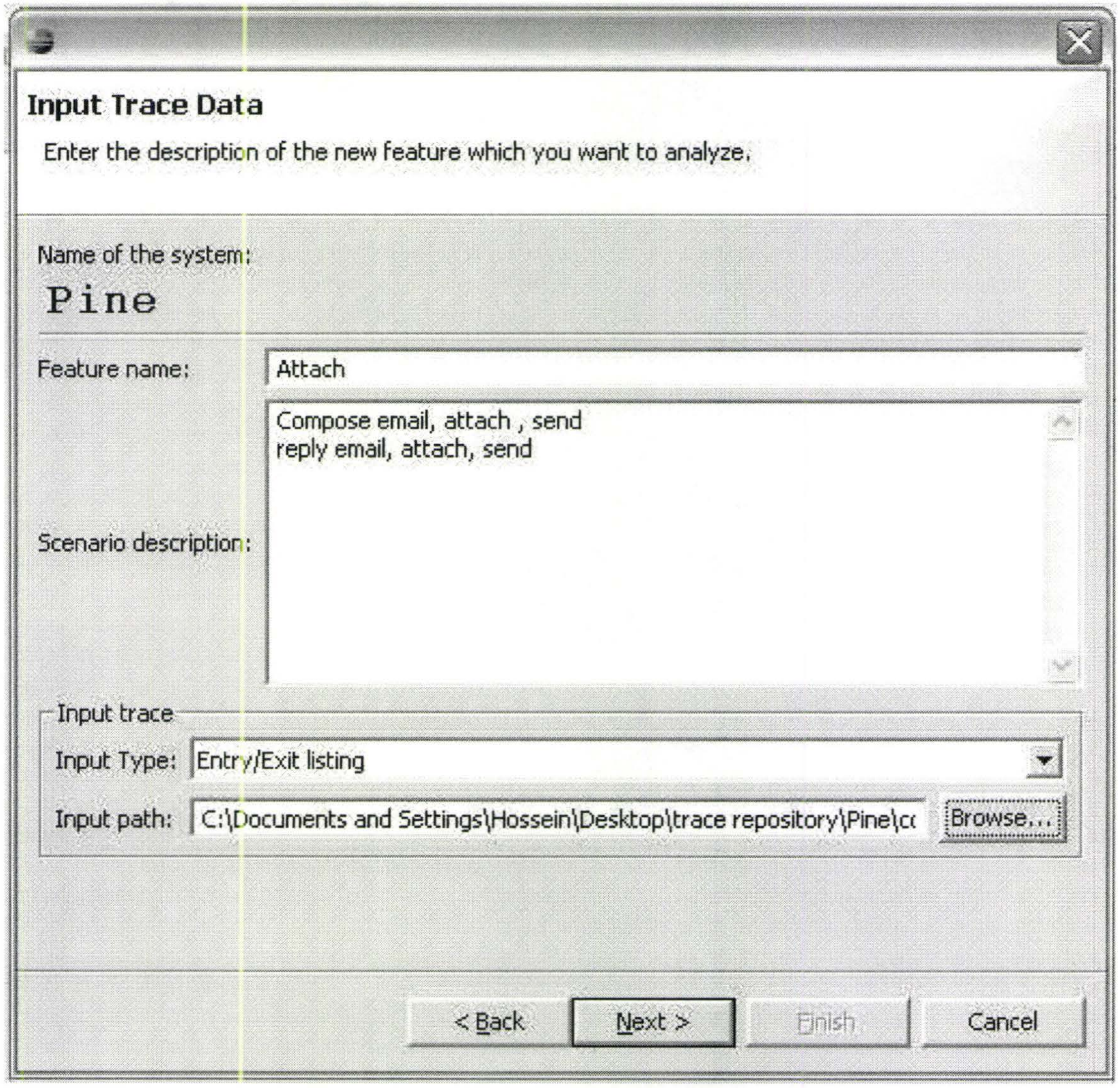
List of analyzed features:

AddressBook
Compose
folderList
messageIndex

Next step

[✓ Add a new feature]
Start Pattern Analysis

Figure A.6: History page, provides a history for the selected system in the Dynamic Alborz Run wizard.



Input Trace Data

Enter the description of the new feature which you want to analyze.

Name of the system:
Pine

Feature name: Attach

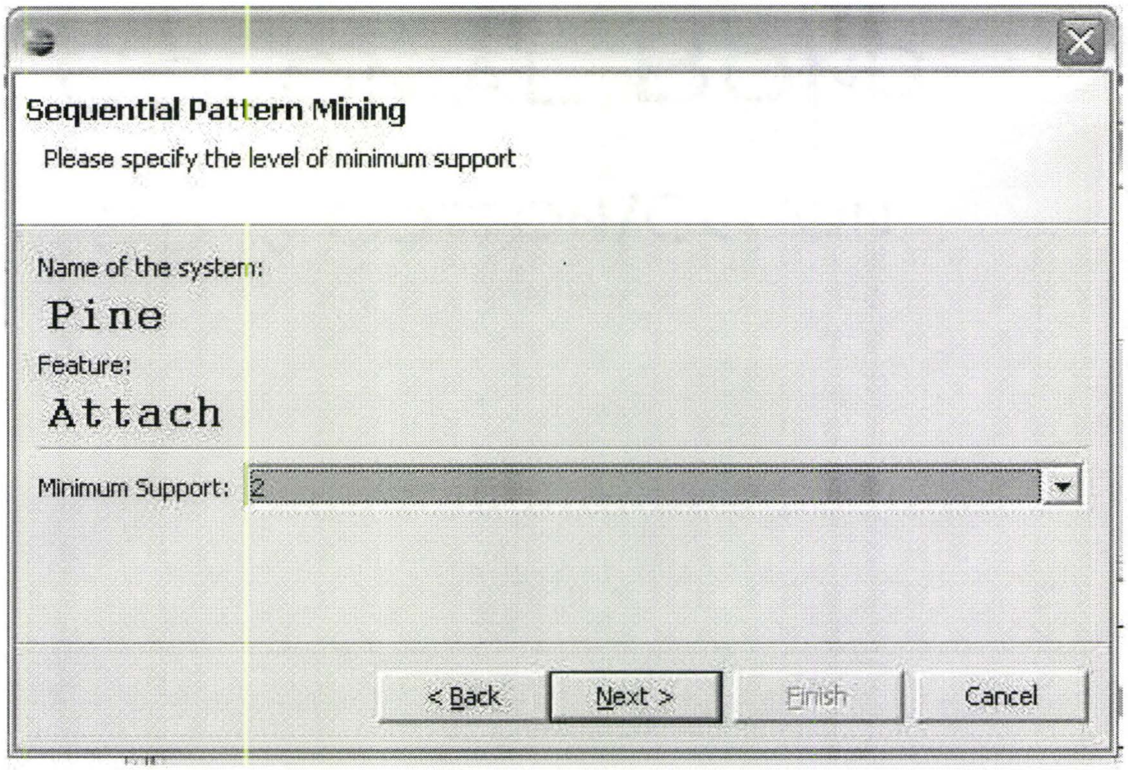
Scenario description:
Compose email, attach , send
reply email, attach, send

Input trace

Input Type: Entry/Exit listing

Input path: C:\Documents and Settings\Hossein\Desktop\trace repository\Pine\cc

Figure A.7: Input page, provides an interface for adding a new feature to the system.


Sequential Pattern Mining

Please specify the level of minimum support

Name of the system:

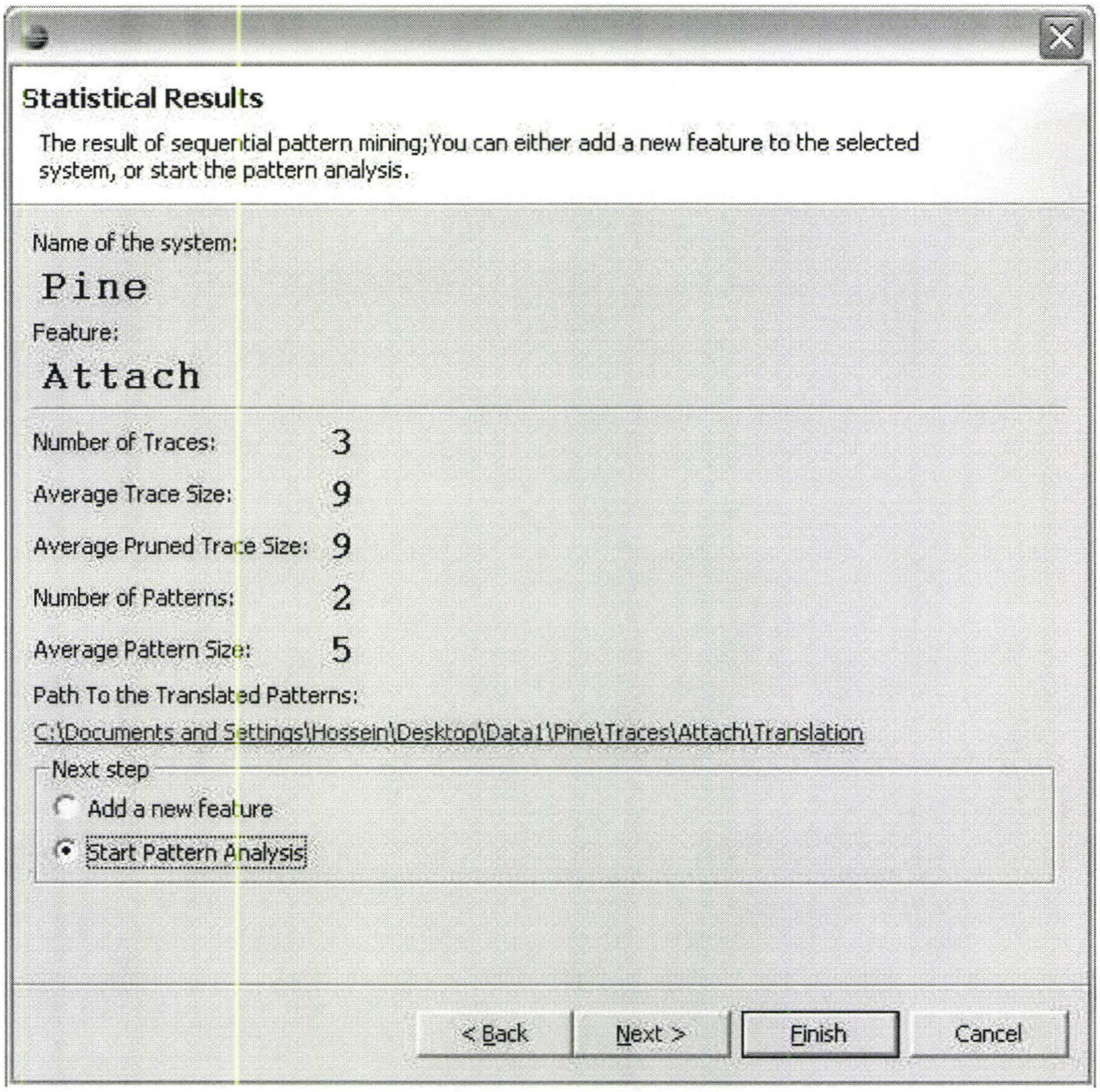
Pine

Feature:

Attach

Minimum Support: 2

Figure A.8: Data Mining page, user can select the minimum support for the data mining operation.



Statistical Results

The result of sequential pattern mining;You can either add a new feature to the selected system, or start the pattern analysis.

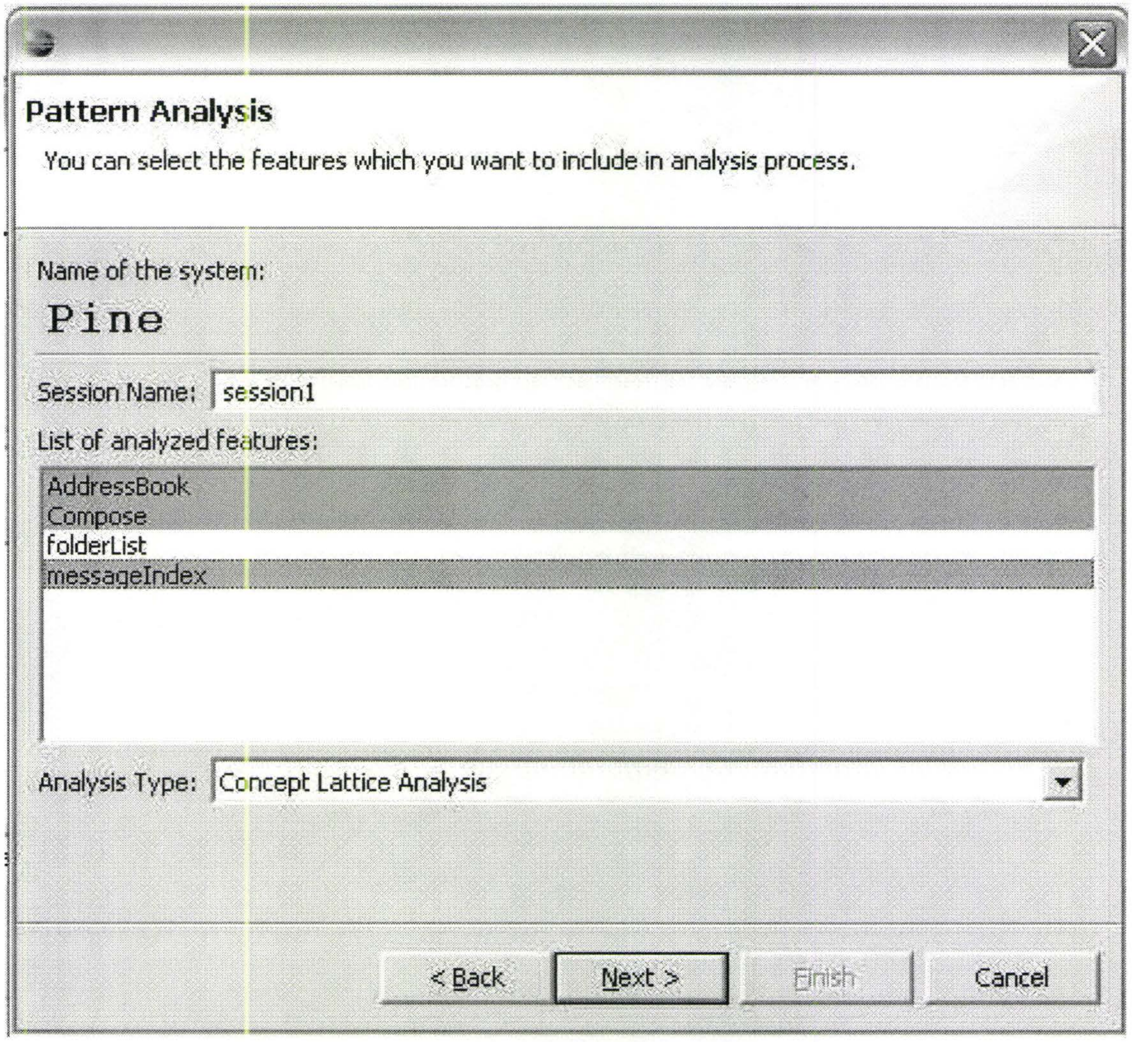
Name of the system:
Pine
Feature:
Attach

Number of Traces:	3
Average Trace Size:	9
Average Pruned Trace Size:	9
Number of Patterns:	2
Average Pattern Size:	5

Path To the Translated Patterns:
C:\Documents and Settings\Hossein\Desktop\Data1\Pine\Traces\Attach\Translation

Next step
Add a new feature [✓ Start Pattern Analysis]

Figure A.9: Statistical Result page, provides statistical information about the extracted execution patterns.



Pattern Analysis

You can select the features which you want to include in analysis process.

Name of the system:

Pine

Session Name: session1

List of analyzed features:

AddressBook
Compose
folderList
messageIndex

Analysis Type: Concept Lattice Analysis

Figure A.10: Pattern Analysis page, provides an interface for selecting the analysis type and the features that should be involved in a specific analysis session.

Bibliography

[1] Xfig version 3.2.3. http://www.xfig.org/.

[2] Pine email client version 4.4.0. http://www.washington.edu/pine/.

[3] Eclipse version 3.0. http://www.eclipse.org.

[4] Formal concept analysis toolkit version 1.0.1. http://sourceforge.net/projects/conexp.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[6] Thoms Bell. The concept of dynamic analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234, London, UK, 1999. Springer-Verlag.

[7] Garrett Birkhoff. *Lattice Theory.* American Mathematical Society, 1st edition, 1940.

[8] Harold W. Cain, Barton P. Miller, and Brian J. N. Wylie. A callgraph-based search strategy for automated performance diagnosis (distinguished paper). *Lecture Notes in Computer Science*, 1900:108+, 2001.

[9] Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[10] Carlos Montes de Oca and Doris L. Carver. A visual representation model for software subsystem decomposition. In *Proceedings of the Working Conference on Reverse Engineering*, pages 231–240, 1998.

[11] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. Technical report, Software Engineering Research Center (SERC), 2004.

[12] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Derivation of feature component maps by means of concept analysis. Fifth European Conference on Software Maintenance and Reengineering, March 2001.

[13] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29:210 – 224, March 2003.

[14] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 447–454, New York, NY, USA, 2002. ACM Press.

[15] M. Ernst. Static and dynamic analysis: synergy and duality, 2003.

[16] Usama M. Fayyad. *Advances in knowledge discovery and data mining.* MIT Press, Menlo Park, Calif., 1996.

[17] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Recognizers for extracting architectural features from source code. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 252–261, Toronto, Canada, July 14-16 1995.

[18] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, 1997.

[19] Alok Mehta and George T. Heineman. Evolving legacy systems features using regression test cases and components. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 190–193, New York, NY, USA, 2001. ACM Press.

[20] Bhatia Nikhil, Moore Shirley, Wolf Felix, Dongarra Jack, and Mohr Bernd. A pattern-based approach to automated application performance analysis. Proceedings of the Workshop on Patterns in High Performance Computing (patHPC 2005), 2005.

[21] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[22] Roger S. Pressman. *Software Engineering, A Practitioner Approach.* McGraw-Hill, third edition, 1992.

[23] Erik Putrycz. Using trace analysis for improving performance in cots systems. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 68–80. IBM Press, 2004.

[24] Tamar Richner and phane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 13, Washington, DC, USA, 1999. IEEE Computer Society.

[25] K. Sartipi, N. Dezhkam, and H. Safyallah. An orchestrated multi-view software architecture reconstruction environment. In *WCRE '06: Proceedings of the Thirteenth Working Conference on Reverse Engineering*, 2006.

[26] Kamran Sartipi. Alborz: A query-based tool for software architecture recovery. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 115–116, Toronto, Canada, May 2001.

[27] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching*. PhD thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2003.

[28] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, Nov./Dec. 1999.

[29] OC Systems. Aprobe version 4.2 for unix, 2003.

[30] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 112–121, 2004.

[31] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the ICSE 1999*, pages 246–255, 1999.

[32] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[33] Felix Wolf, Bernd Mohr, Jack Dongarra, and Shirley Moore. Efficient pattern search in large traces through successive refinement. In *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, Pisa, Italy, 2004.

[34] Kenny Wong. Software understanding through integrated structural and run-time analysis. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 73. IBM Press, 1994.

[35] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 134–142, Washington, DC, USA, 2005. IEEE Computer Society.

[36] Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 329, Washington, DC, USA, 2004. IEEE Computer Society.