Dynamic Knowledge Extraction from Software Systems using Sequential Pattern Mining

Kamran Sartipi  
Department of Computing and Software, McMaster University  
Hamilton, Ontario, L8S 4K1, Canada  
sartipi@mcmaster.ca  
http://www.cas.mcmaster.ca/~sartipi

Hossein Safyallah  
Department of Computing and Software, McMaster University  
Hamilton, Ontario, L8S 4K1, Canada  
safyalha@mcmaster.ca  
http://www.cas.mcmaster.ca/~safyalh

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 software 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 paper presents a novel technique for dynamic analysis of software systems to identify the implementation of certain software functionality 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 functionality. 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 toolkit and experimented with two case studies Xfig drawing tool and Pine email client with very promising results.

Keywords: Sequential Pattern Mining; Feature Extraction; Execution Trace; Scenario Analysis.

1. Introduction

There is a growing attention towards the dynamic aspects of software systems as a challenging domain in the software reverse engineering [27, 14]. 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 [25], software execution visualization [23], and feature to
code assignment [13], 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 which is tedious and is not applicable
in many cases. In this paper, we propose a novel approach to dynamic analysis of
software systems, in order to identify the implementation of the software features
without any prior knowledge about its source code implementation. In this context,
dynamic analysis is performed by executing a group of well-defined task scenarios on
the software system and by analyzing the execution results. Dynamic analysis with
its characteristics to extract system functionality has several challenges compared
to the 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 a set of task scenarios with a specific shared feature and
execute them on the software system in order to generate execution traces. The ap-
plication of a sequential pattern mining algorithm on the extracted execution traces
allows us to obtain highly frequent sequence-patterns (or 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 of a certain software feature (i.e., the group
of feature-specific functions), we assess the impact of the feature on a portion of soft-
ware 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 cohe-
sion (“relative functional strength of a module” [24]) 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 [29].

This paper has been organized as follows. Related work is discussed in section
2. Section 3 briefly presents the proposed framework. Section 4 provides formal
definitions for the proposed approach. Sections 5 to 7 discuss three stages of the
proposed framework, as: execution trace extraction, execution pattern mining, and
pattern analysis, respectively. Section 8 provides an overview of the proposed struc-
tural evaluation metrics. Section 9 presents the results of experimentation on Xfig
drawing tool and Pine email system. Finally, section 10 concludes the paper and
provides guidelines for the future research.
2. Related Work

In this section, we briefly present the approaches in dynamic analysis of a software system that relate to our work. First, we describe the approaches in software reverse engineering that employ data mining techniques. Then, existing approaches to application of concept lattice analysis in this field is discussed. Finally, we present recent approaches in dynamic analysis of software systems. Due to space limitation, it is not possible to present all important related approaches in different fields covered by this paper.

In dynamic analysis of software systems, El-Ramly et al. [15] applied a sequential pattern mining technique to identify 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 [38] a web-mining technique is applied on program dynamic call graphs, where nodes represent classes and edges represent method invocation. In this approach, 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. In this paper we use data mining algorithm sequential pattern mining in order to extract frequent patterns of function calls.

Concept lattice analysis provides a visualization means to identify maximum-size groups of objects that have common attributes [17]. 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 [32, 21, 34], 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, Koschke and Simon [13, 14] 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 are considered as the attributes 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. Tonellan et al. [33] applied concept lattice analysis on execution traces of a software system to mine the potential program-aspects that exist in the software.

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 [8, 16] 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. In an approach to software understanding using execution traces Pauw et al. [23] 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. [39] used execution traces as clues for tracing the evolution of a software system. In [40] 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 [22, 36, 10] performance analysis of parallel systems is studied by using execution traces of the software systems. In [10, 36] a program’s execution trace is searched for certain predefined patterns that indicate inefficient behavior. In [12] 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. N. Wilde et al. [35] 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.

A major challenge in the trace-based dynamic analysis approaches would occur right at the beginning of the analysis, that is managing very large traces [20, 30, 39]. Hamou-Lhadj and Lethbridge [19] provide a framework to compress the execution traces by removing loop-based redundancies, where the process is reversible. The method is based on identifying identical sub-trees in the dynamic tree that is generated from an execution trace. We also use a similar technique to remove the loop-related redundancies. Reiss and Renieris [26] propose a set of trace compaction techniques including string compaction, dynamic call graph analysis, grammar-based encoding and finite state automata. Greevy and Ducasse [18] extract execution traces to generate a mapping between software features and classes by comparing the classes that generate the execution trace for different features.

Our approach in this paper exploits an 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.

3. Proposed Framework

Figure 1 illustrates different steps of the proposed framework for assigning software features onto its source code. The framework provides 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 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 remaining of this section these stages are briefly
Fig. 1. Proposed framework for identifying the implementation of the functional aspects of a software system in the source code as a means to measure the structural impacts of different software features.

described.

Stage 1. 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 software 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 move a figure on the computer screen would constitute such a feature-specific scenario set. In the next step, the software under study is instrumented\(^a\) 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 pre-processing step we transform the extracted entry/exit listing into a sequence of function invocations and also remove all redundant function calls caused by the cycles of the program loops. The pruned execution traces are then fed into the execution pattern mining engine in the next stage. The pre-processing operation will be discussed in more details in Section 5.

\(^a\)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.
Stage 2. 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. Each execution pattern is a candidate group of functions that implement a common feature within a scenario set. 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 6.

Stage 3. Execution pattern analysis: even in one feature-specific scenario set, 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 (the latter is not discussed in this paper). We use concept lattice analysis to cluster the group of functions in extracted patterns that exclusively correspond to a shared feature of a scenario set, and also to cluster the group of functions in patterns that are common to every scenario set. This stage is discussed in Section 7.

Stage 4. Structural evaluation: in a further operation, by associating the feature-specific functions to the system’s structural modules (i.e., files of the system) two metrics are defined, namely structural cohesion and functional scattering that together provide a means for measuring the impact of individual features on the structure of the software system.

4. Formal definition of the approach

In this section, we define the common terminology that we use throughout this paper to describe the execution pattern mining and pattern analysis aspects of the proposed approach.

We use Z notation [37] to formally define the concepts in this section. In the Z notation, a “set” can be defined as \( \{ D \mid P \bullet E \} \) denoting a set of values consisting of all values of the term \( E \) for the declared variables in \( D \) that satisfy the constraint \( P \). The predicate \( P \) and term \( E \) contain the free variables defined in \( D \). For example, \( \{ x : \mathbb{N} \mid x \leq 5 \bullet x^2 \} \) denotes the set \( \{1, 4, 9, 16, 25\} \). The term \( E \) and its preceding “heavy dot” can be omitted which results \( \{ x : \mathbb{N} \mid x \leq 5 \} = \{1, 2, 3, 4, 5\} \).

The existential quantifier “\( \exists \)” is used to define a new variable. The general form of the existential quantifier is \( \exists D \mid P \bullet Q \) where \( D \) represents declarations, \( P \) represents a predicate acting as the constraint and \( Q \) represents the predicate being quantified. The constraint bar “\(|\)” and the constraining predicate \( P \) can be omitted, which results: \( \exists D \bullet Q \).

The universal quantifier “\( \forall \)” is used to define all variables that have certain properties. The general form of the universal quantifier is \( \forall D \mid P \bullet Q \). The constraint bar “\(|\)” and the constraining predicate \( P \) can be omitted, which results: \( \forall D \bullet Q \).
• Feature $\phi_i$ corresponds to a single functionality (number $i$) that is provided by the subject software system; and $\Phi$ is the set of all available features.

• Scenario $s_j$ is a sequence of features $\phi_i \in \Phi$; hence $s_j = [\phi_{j1}, \phi_{j2}, \ldots, \phi_{jn}]$. Also $\mathcal{S}$ is the set of all applicable scenarios on the system.

• Feature-specific scenario set $S_{\phi_i}$ is a set of scenarios $s_j$’s that use specific feature $\phi_i$; such that $S_{\phi_i} = \{s_j \mid s_j \in \mathcal{S} \land \phi_i \in s_j\}$.

In this model, the execution of a scenario $s_j$ on the subject software system is represented by a traversal of the software’s static call graph, where each tree traversal generates a dynamic call tree $dct_j$ that is defined below.

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

• Dynamic Call Tree $dct_j = \langle \mathcal{F}', E \rangle$ is a tree where the set of nodes $\mathcal{F}'$ represents different invocations\(^b\) of functions $f \in \mathcal{F}$, and $E \subset \mathcal{F}' \times \mathcal{F}'$ represents the set of edges among function invocations.

• Dynamic Software System $\Psi$ models the subject software system as a set of all possible dynamic call trees $dct_j$’s that are generated by the execution of task scenarios $s_j \in \mathcal{S}$ on the subject software system. We also model a scenario execution $\mathcal{E}(s_j)$ on the subject software system as a look up operation which returns the corresponding dynamic call tree of the scenario $s_j$, hence $\mathcal{E} : \mathcal{S} \rightarrow \Psi$.

For simplicity and without loss of generality, we use $\Psi$ to represent only dynamic call trees that correspond to a group of $k$ feature-specific scenario sets that are represented by “the restricted $\mathcal{S}$” as $\mathcal{S} = \bigcup_{i=1}^{k} S_{\phi_i}$. This means the subject software system $\Psi$ can only execute the intended $k$ scenario sets, not all possible scenarios.

• Preprocessor $\Pi : \Psi \rightarrow \mathcal{T}$ is a tree pruning and serialization operation, where: i) replaces multiple instances of identical subtrees (i.e., repeated under a particular parent node) in a dynamic call tree $dct_j$, with one of those instances; and ii) maps the loop-free $dct_j$ to an execution trace $t_j \in \mathcal{T}$ using a depth first traversal operation on the dynamic tree $dct_j$, where the sequence of visited nodes in this traversal constitute execution trace $t_j$. In this form, $\mathcal{T}$ represents the set of all traces $t_j$ that are stored in a repository to be used for execution pattern generation process; and $\mathcal{T}_{\phi_i}$ represents feature specific traces that correspond to scenario set $S_{\phi_i}$. Hence:

$$
\mathcal{T}_{\phi_i} = \{t \mid \forall s_j \in S_{\phi_i}, \exists t = \Pi(\mathcal{E}(s_j))\}
$$

$$
\mathcal{T} = \bigcup_{i=1}^{k} \mathcal{T}_{\phi_i}
$$

• Execution pattern $p_x$ is defined as a subsequence of a trace $t$ (i.e., a contiguous sequence of functions $f$ within trace $t$) that is supported\(^c\) by at least

\(^b\)In this context, two different invocations of a single function $f \in \mathcal{F}$ are represented as $f^i, f^j \in \mathcal{F}'$ ($i \neq j$). \(^c\) $p_x$ exists in certain number of execution traces $t_j$’s, where the collection of $t_j$’s are called the
MinSupport number of execution traces in $T$; where the support of execution pattern $p_x$ is defined below. Also, $P$ is the set of all execution patterns $p_x$.

$$\text{sup}(p_x, T) = \{ t \mid t \in T \land \exists i \geq 0 \land \forall j \ (i \leq j < (i + |p_x|) \rightarrow p_x[j - i] = t[j]) \}.$$  

- **Execution pattern miner** $\Upsilon(T, n)$ is a function which receives the set of execution traces $T$ and MinSupport $n$, and returns all execution patterns $p_x \in P$ that exist in at least $n$ execution traces $t$’s in $T$.

$$\Upsilon(T, n) = \{ p \mid \text{subSeq}(p, t) \land t \in T \land |\text{sup}(p, T)| \geq n \} \equiv P.$$  

Depending on the level that functions are participated in execution patterns of different feature-specific scenario sets, we define two categories of functions: feature-specific functions and common functions as follows.

- $F_{\phi_i}$ is a set of feature-specific functions that are used to implement specific feature $\phi_i$.

$$F_{\phi_i} = \{ f \mid f \in p \land p \in \Upsilon(T_{\phi_i}, n) \land n = |S_{\phi_i}| \}$$

- $F_{\text{com}}$ is the set of common functions that exist in the extracted patterns of almost every feature-specific scenario set.

$$F_{\text{com}} = \{ f \mid f \in p \land p \in \Upsilon(T, n) \land n \approx \sum_{i=[1..k]} |S_{\phi_i}| \}$$

Based on the above definitions, we present the details about different stages of the proposed dynamic analysis framework in Section 3. These activities are summarized below.

1. Execute $k$ different feature-specific scenario sets $S_{\phi_i}$ ($i \in [1..k]$) on the subject software system and generate the corresponding dynamic call trees $\Psi$.
2. Preprocess the dynamic call trees in $\Psi$ in order to eliminate the loop-based repetitions and generate feature specific execution traces $T_{\phi_i}$ (and consequently all execution traces $T$).
3. Apply execution pattern miner on $T_{\phi_i}$’s ($i \in [1..k]$) to extract $k$ sets of feature-specific patterns $\Upsilon(T_{\phi_i}, n)$.
4. Apply concept lattice analysis on functions of different patterns in all $\Upsilon(T_{\phi_i}, n)$’s in order to separate feature-specific functions $F_{\phi_i}$ from common functions $F_{\text{com}}$.
5. Finally, study the impact of the implemented features on the structure of the system.

support of the pattern $p_x$.  

8
5. Execution trace extraction (stage 1)

In order to run different scenario sets on the subject software system, we need to instrument the system. We adopt Aprobe [4] which is a binary level software instrumentation tool to insert patches, namely probes, within the binary image of the executable program. We use a pre-defined probe (namely trace) which generates text messages at both entrance and exit of each function. Consequently by running the selected feature-specific scenario sets we obtain entry/exit listings that are transformed into dynamic call trees in a further step. For space limitation this transformation is not discussed in this paper. This step that represents $E : S \rightarrow \Psi$ generates different groups of dynamic call trees corresponding to different scenario sets that should be pre-processed and converted to the execution traces for the execution pattern analysis.

Pre-processing. The preprocessor $\Pi : \Psi \rightarrow T$ is a dynamic call tree pruning and trace serialization operation. 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 dynamic analysis. The effective trace of functions corresponding to a scenario execution 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 call sequence. In this work, we ignore recursive function traces and focus on pruning the loop-based redundancies.

We transform the entry/exit listing that is generated by executing a task scenario on the software system into a dynamic call tree where 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. We also assign an integer ID to each tree node, where roots of identical subtrees possess identical IDs. This technique significantly simplifies the task of localizing and eliminating the loop-based redundancies at proper children of each node in the dynamic call tree. Therefore, the loop redundancy removal problem is reduced to identification of identical subtrees that are repeated under a particular node.

Figure 2(a) shows Procedure $Foo$ that produces a long trace with repetitions of “$F1, F2$”. Figure 2(b) illustrates a small portion of a dynamic call tree that is generated from an execution of the Procedure $Foo$. Furthermore, each node in this dynamic call tree is annotated with its ID. Note that functions $F1$ and $F2$ are called several times by function $Foo$, hence they acquire different IDs depending on their run-time behavior.

The pruning process is as follows: i) Generate a string representation of ID’s from different sibling subtrees. ii) Apply a repetitive string finder algorithm (Crochemore [11]) to transform the original string (with repetitions) into a new string with no repetitions. iii) In the new string, each group of repetitions is shown as one instance
Fig. 2. (a) A dummy procedure which generates loop-based repetition. (b) Dynamic call tree generated from an execution of Procedure Foo.

Fig. 3. (a) A string containing repetitions. (b) Representation of (a) in the form of one instance of string repetition. (c) Another representation of (a) in the form of one instance of string repetition.

of the repetition that is labeled with the number of the repetitions.

For example, in Figure 3(a) the string $F1,F2,F1,F2, \ldots, F1,F2$ is transformed into $(F1,F2)^n$ in Figure 3(b). There may exist more than one pattern of repetitions for a given string (e.g., strings in Figures 3(b) and (c)) 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 corresponding 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.

6. Execution pattern mining (stage 2)

In the data mining literature, sequential pattern mining is used to extract frequently occurring contiguous (or interleaved) sequences of events (known as patterns) among the sequences of customer transactions [7]. In the proposed approach, we use a modified version of the original sequential pattern mining algorithm by Agrawal [7], where an execution pattern is a contiguous part of an execution trace that is supported by $MinSupport$ number of the execution traces.

The execution pattern miner $\Upsilon(T_{\phi_i}, n)$ receives different sets of pruned (loop-free) feature-specific execution traces $T_{\phi_i}$ as well as the MinSupport $n$ for each set, and generates different sets of execution patterns containing both feature-specific functions $F_{\phi_i}$ and common patterns of functions $F_{com}$.

Figure 4(a) illustrates the application of pattern miner $\Upsilon(T_{\phi_i}, n)$ on three sets of
feature-specific execution traces. As a result, feature-specific and common patterns are extracted. The noise execution patterns in Figure 4(a) are a kind of common patterns that are not as frequent as common patterns, and hence are not as important as the two main types of patterns. Each feature-specific pattern only exists in its corresponding traces; common patterns almost exist everywhere; and noise patterns, are those whose frequencies are in the middle. In this example, the functions that implement each feature are highlighted; also the separation of different types of patterns is trivially feasible for human. However, for large trace sizes a large number of patterns of different types are generated whose separation is almost impossible by human’s inspection.

The pattern miner generates a large number of execution patterns such that the majority of these patterns are redundant sub-patterns of a large execution pattern and cause size explosion that significantly increase the computational complexity. In order to identify and eliminate the sub-patterns of a final execution pattern, we use a Trie data structure\(^d\) and annotate its nodes with the function names. Figure 4(b) illustrates a Trie data structure that we use for representing the trace of pattern functions. Each leaf Trie node can be labeled as final (i.e., an execution pattern) or subpattern (a redundant pattern) where the latter will be eliminated. In doing so, the sequence of functions in each execution pattern \(p_x\) is stored along a path from the root to the leaf of the Trie, and the corresponding leaf is marked “final” if the sequence does not already exist in the Trie.

### 7. Execution pattern analysis (stage 3)

We employ a strategy to focus on execution patterns corresponding to a specific feature within each group of scenario set as well as common patterns that exist in

\(^d\)A Trie is a binary search tree data structure that stores the information about the contents of each node in the path from the root to the node.
Almost all scenario sets. In the following, the different kinds of functions that exist in extracted execution patterns along with the corresponding extraction mechanisms are presented.

**Feature-specific functions** $\mathcal{F}_{\phi_i}$. Feature-specific functions are core functions that implement the targeted feature $\phi_i$ of a feature-specific scenario set $S_{\phi_i}$. In order to extract feature-specific functions, we should increase the level of MinSupport $n$ of the pattern miner $\Upsilon(T_{\phi_i}, n)$ to a number that covers the majority of the scenarios in $S_{\phi_i}$.

**Common functions** $\mathcal{F}_{\text{com}}$. Common functions exist in almost every task scenario of the software system, e.g., software initialization / termination operations, or mouse tracking. Such functions exist in every execution trace of every scenario-set $S_{\phi_i}$. Therefore, it is extracted along with the feature-specific function mentioned above. 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 order to extract such functions, we should use a filtering mechanism such as concept lattice analysis to filter out the feature-specific functions from this group of functions.

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. In the rest of this section, we present a filtering mechanisms to separate the common functions from feature-specific functions.

### 7.1. Concept Lattice Analysis

We employ the visualization power of mathematical concept analysis (Birkhoff 1940 [9] and Ganter & Rudolf 1999 [17]) as a strategy to cluster the group of functions in execution patterns that either: i) exclusively correspond to a shared feature of a scenario set; or ii) shared among all scenario sets. A *formal context* is defined as a triple $\mathcal{C} = (O, A, \mathcal{R})$ which represents the relation $\mathcal{R}$ between objects $O$ and their
attribute values $\mathcal{A}$. A \emph{concept} $c_x$ is a maximal collection of objects sharing maximal common attribute-values. Figure 5 illustrates three steps for defining the set of concepts: (a) context table among objects and their attributes, where maximum groups of shared features among objects can be identified as maximum group of $X$’s; (b) concept lattice, where each node represents a concept $c_x$; and (c) the list of concepts, where each concept $c_x$ is represented by as a tuple of \texttt{extent} “$\text{Ext}(c_x)$” and \texttt{intent} “$\text{Int}(c_x)$”. A concept lattice has the following characteristics:

- Each lattice node (i.e., a concept) may have labels for 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 below it in the lattice (directly below or separated by some links).

In our approach, we present the relation $\mathcal{R}'$ between “set of features $\Phi$” and “set of functions $\mathcal{F}$” such that $\mathcal{C} = (\Phi, \mathcal{F}, \mathcal{R}')$. In other words, an object is a targeted feature $\phi_i \in \Phi$ of a feature-specific scenario set $S_{\phi_i}$, and an attribute is a function $f$ that participates in the execution patterns corresponding to $S_{\phi_i}$. Applying concept lattice analysis to the proposed formal context will result in separation of “feature-specific functions $\mathcal{F}_{\phi_i}$” from “common functions $\mathcal{F}_{\text{com}}$” as follows.

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. We exploit this property to group functions of the extracted execution patterns. In our case, common functions $\mathcal{F}_{\text{com}}$ are executed through almost every task scenario of the software system; hence these functions cluster in upper region of the lattice. However, this property also prevents us from distinguishing different groups of functions that implement different common functionality. On the other hand, feature-specific functions $\mathcal{F}_{\phi_i}$ are located in the lower region of the lattice. Consequently, functionality of these functions can be easily identified using the meaning of the specific feature of the corresponding lattice node. Particularly, a concept whose extent consists of a single object (feature $\phi_i$) collects all functions that exclusively implement that feature.

8. Structural Evaluation of Software System

Software systems are continuously evolving throughout their life time 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 dam-
age, high maintenance cost, and eventually retirement of a legacy system. To help this situation, the task of the software maintainers is to measure the impair on the structure of the software system and assess the current state of the resulting legacy 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 section, we provide two metrics to assess the structural merit of the software system, namely: functional scattering and structural cohesion. The proposed functional scattering metric examines the distribution of a set of functions that implement a family of relevant features (could consist of one single 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 relatedness of the functions that reside in each structural unit (module). This measure of cohesion is much closer to the original definition of cohesion (“relative function strength of a module” [24]) than using static structural techniques such as inter-/intra-edge connectivity of the components.

A feature family \( \Phi_i \) (also denoted as a logical module) is a set of related features that possess similar functionality with regard to a single feature \( \phi_i \), i.e., they share feature-specific functions from \( \mathcal{F}_{\phi_i} \). In order to measure the functional scattering of a logical module \( \Phi_i \), we assess the level of distribution of the functions that correspond to \( \Phi_i \) (denoted as \( \mathcal{F}_{\Phi_i} \)) over the whole structure of the relevant system files. Similarly, we compute the structural cohesion of a system file as the level of relatedness of its functions with respect to the functions in a logical module i.e., \( \mathcal{F}_{\Phi_i} \). To obtain these metrics, first we identify and collect the functions of the logical module using the above discussed concept lattice analysis. Then, the source files that contain these functions are identified and the ratio of shared functions over the total functions are calculated as follows.

- Let \( \mathcal{F}_l \) be the set of functions that are defined in file \( l \).
- Let \( L_{\Phi_i} = \{l_1, l_2, \ldots, l_k\} \) be the set of system files that contain all functions of the logical module \( \Phi_i \) (i.e., \( \mathcal{F}_{\Phi_i} \)).
- Structural cohesion of file \( l \) with respect to logical module (feature family) \( \Phi_i \) is defined as:

\[
SC_{\Phi_i}(l) = \frac{|\mathcal{F}_l \cap \mathcal{F}_{\Phi_i}|}{|\mathcal{F}_l|}
\]

- Functional scattering of logical module (feature family) \( \Phi_i \) is defined based on
the distribution of functions in $F_{\Phi_i}$ over files in $L_{\Phi_i}$ as:

$$\text{FS}(\Phi_i) = 1 - \frac{\sum_{l \in L_{\Phi_i}} \text{SC}_{\Phi_i}(l)}{|L_{\Phi_i}|}$$

A software system with high structural cohesion $\text{SC}_{\Phi_i}(l)$ for its individual files and low functional scattering $\text{FS}(\Phi_i)$ among its files represents a modular system that requires low maintenance effort. However, a high degree of functional scattering corresponding to a feature family $\Phi_i$ directly signifies a high structural impair that is caused by that feature family. Hence the system requires more maintenance effort to tackle with the consequences of propagated change to other software files.

9. Experiments

In this section, we apply the proposed dynamic analysis technique on two medium-size open source systems. The developed dynamic analysis tool is an Eclipse plug-in [2] and has been developed as an extension to the Alborz reverse engineering toolkit [31] to enhance the scope of Alborz to cover both static and dynamic analysis of a software system.

9.1. 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 application of the proposed dynamic analysis technique on the Xfig drawing system, according to the stages of the proposed framework in Section 3. Figure 6(a) illustrates the group of task scenarios that form a feature specific scenario set, where the flip operation is the specific feature. 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. We apply the above strategy to generate feature-specific scenario sets that each target one feature within Figure 7. We execute the scenarios of each feature-specific scenario set \( S_\phi \) on the instrumented Xfig system and obtain the corresponding entry/exit listings. After pruning the extracted entry/exit listings from loop-based function calls we apply the execution pattern mining process to obtain the patterns of function call sequences. Figure 7 presents the statistics about attributes of a group of feature-specific scenario sets that we used in analyzing Xfig. 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).

**Concept Lattice Analysis.** We supply the resulting execution patterns of the Xfig experiments to a concept lattice generation tool (concept explorer [3]) in order to view the distribution of the feature functions on the lattice. As it was discussed in Section 7.1 the feature-specific functions are clustered around the nodes (concepts) that each represent a specific feature. Similarly, the common function that are shared among a majority of concepts are clustered at the upper region of the lattice, and hence their common operations can not be distinguished from each other. The visualization power of the concept lattice will also allow us to cluster the group the functions of highly related features (i.e., lower region lattice nodes) into feature families where they present similar behaviors. In Figure 6(b) 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. On the other hand, the upper nodes collect those functions of Xfig corresponding to common patterns, such as: software initialization and termination, mouse pointer handling, canvas view updating, and side ruler management.

**Structural Evaluation.** Based on inspecting the source files of Xfig, we measure the structural cohesion of corresponding source files, as well as the functionality scattering of the feature families under study. The results of this evaluation for three feature families Draw Ellipse, Copy, and Draw Spline are presented in Figure 8.

For the three mentioned feature families we inspect the Xfig source files that
define the functions that implement the corresponding logical module $F_{\Phi_i}$ of that feature family. The results of measuring the structural cohesion $SC_{\Phi_i}(l)$ of these files are presented in Figure 8. These results indicate that file $d_{\text{ellipse}}$ has high cohesion with respect to logical module of feature family $\text{Ellipse}$; files $e_{\text{copy}}$, and $e_{\text{move}}$ are also highly cohesive with respect to feature family $\text{Copy}$; and finally, file $d_{\text{spline}}$ is cohesive with respect to feature family $\text{Spline}$. However, study of the functional scattering measures allows us to better interpret the characteristics of these logical modules. For example, in the case of $\text{Ellipse}$ a small portion of the logical module $F_{\Phi_i}$ is located in a large file $u_{\text{elastic}}$ which results in a high functional scattering measure. Whereas, in the case of $\text{Copy}$ feature family, the logical module almost covers two files $e_{\text{copy}}$ and $e_{\text{move}}$ which indicates low scattering.

In the case of $\text{Spline}$, the logical module is almost equally scattered among four files each covering a small portion of the files and hence indicating high functional scattering. We also adopt a minimum threshold value of 10% in order to consider a file in the calculation of the above measurements. The results in Figure 8 are promising in the sense that they reflect meaningful measures with respect to the sizes of logical modules and system files shown. Regarding the results of our structural evaluations, we can predict high maintenance activities regarding any change to the feature families $\text{Ellipse}$ and $\text{Spline}$. However, changes to the $\text{Copy}$ feature family would not propagate throughout the system which indicates less maintenance activity is required.

### 9.2. Dynamic Analysis of Pine Email Client

Pine 4.4.0 [6] is an open source, large-size (207 KLOC), C language email client for reading, sending, and managing electronic messages. For this case study, we repeated exactly the same steps we discussed in the study of the Xfig drawing tool. We examined 4 different features of Pine for: composing emails, managing the folder lists, address book, and message index. Figure 9 presents the result of execution traces extraction as well as execution pattern mining for the above 4 Pine features. By repeating this process and targeting other features of the system with
proper sets of scenarios, we could incrementally explore the Pine system's overall functionality. By spreading the extracted execution patterns over a concept lattice we could separate feature-specific functions from common functions that implement experimented features. Finally, based on inspecting the source code of Pine, we measured the distribution of functions that implement each examined feature over the structural units. The results are shown in Figure 10.

For each feature family $\Phi_i$ in Figure 9 we inspected the Pine source files that define the functions that implement the corresponding logical module $F_{\Phi_i}$ of that feature family. The results of measuring the structural cohesion $SC_{\Phi_i}(l)$ of these files are presented in Figure 10. These results indicate high degree of scattering among the examined feature families of Pine. Files context, bldaddr, and reply have low cohesion with respect to logical module of feature family Compose; file send shows high cohesion with respect to feature family Compose. However, study of the functional scattering measures allows us to better interpret the characteristics of these logical modules. For example, in the case of Compose a portion of the logical module $F_{\Phi_i}$ is located in a large file send which results in a high functional scattering

Fig. 8. Structural cohesion and functional scattering measures for three different feature families of the Xfig (the threshold value for this calculations are chosen as 10%).

```
| Feature Family $\Phi_i$ | Contributed File (l) | $|F_l|$ | $|F_l \setminus F_{\Phi_i}|$ | Structural Cohesion $SC_{\Phi_i}(l)$ | Functional Scattering $FS(\Phi_i)$ |
|-------------------------|----------------------|--------|-----------------|------------------|------------------|
| Ellipse                 | ellipse.c            | 16     | 12              | 80%              | 57%              |
|                        | compose.c           | 5      | 2               | 40%              | 32%              |
| Copy                   | context.c           | 4      | 3               | 75%              | 32%              |
|                        | bldaddr.c           | 19     | 15              | 66%              | 66%              |

Ellipse: $\Phi_i$ File (l) = ellipse.c
Compose: $\Phi_i$ File (l) = compose.c
Copy: $\Phi_i$ File (l) = context.c, bldaddr.c
Spline: $\Phi_i$ File (l) = context.c, bldaddr.c, spline.c, spline2.c
Ellipse: $\Phi_i$ File (l) = ellipse.c
Copy: $\Phi_i$ File (l) = context.c, bldaddr.c
Spline: $\Phi_i$ File (l) = context.c, bldaddr.c, spline.c, spline2.c
```

Fig. 9. The result of execution trace extraction and execution pattern mining for a collection of 4 different Pine features.

```
<table>
<thead>
<tr>
<th>Specific Feature of Pine</th>
<th># Different Scenarios</th>
<th>Average Trace Size</th>
<th>Pruned Trace Size</th>
<th>Number of Extracted Patterns</th>
<th>Average Pattern Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compose</td>
<td>8</td>
<td>98381</td>
<td>21920</td>
<td>95</td>
<td>172</td>
</tr>
<tr>
<td>Folder List</td>
<td>3</td>
<td>60553</td>
<td>14906</td>
<td>52</td>
<td>341</td>
</tr>
<tr>
<td>Message Index</td>
<td>5</td>
<td>69741</td>
<td>16229</td>
<td>74</td>
<td>274</td>
</tr>
<tr>
<td>Address Book</td>
<td>3</td>
<td>599221</td>
<td>16024</td>
<td>71</td>
<td>212</td>
</tr>
</tbody>
</table>
```

Fig. 10. Structural cohesion and feature functional scattering measures for four different features the Pine email client (the threshold value for this calculations are chosen as 10%).

```
| Feature Family $\Phi_i$ | Contributed File (l) | $|F_l|$ | $|F_l \setminus F_{\Phi_i}|$ | Structural Cohesion $SC_{\Phi_i}(l)$ | Functional Scattering $FS(\Phi_i)$ |
|-------------------------|----------------------|--------|-----------------|------------------|------------------|
| Compose                 | context.c           | 13     | 2               | 15%              | 74%              |
|                        | bldaddr.c           | 68     | 57              | 54%              | 44%              |
|                        | reply.c             | 65     | 12              | 19%              | 74%              |
| Folder List             | address.c           | 121    | 15              | 17%              | 84%              |
|                        | addtext.c           | 58     | 21              | 14%              | 40%              |
| Address Book            | addtext.c           | 58     | 21              | 14%              | 40%              |
| Message Index           | pine/mailview.c     | 126    | 21              | 17%              | 84%              |
```
10. Conclusion and Future Work

In this work, we proposed a novel approach to dynamic analysis and structural assessment of a software system that takes advantage of repeated patterns of execution traces that exist within the executions of a set of carefully designed task scenarios. The proposed approach takes advantage of techniques such as: execution trace manipulation; sequential pattern mining; string processing; and software visualization through concept lattice analysis. This work benefits from the discovery nature of data mining techniques and concept lattice analysis to extract both feature specific and common groups of functions that implement important features of a software system. The resulting execution patterns provide discovery of valuable information out of noisy execution traces. This technique is centered around a set of task scenarios that share a specific system feature. 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 systems: an interactive drawing tool, and an email client with very promising results in extracting both feature specific and common functions. Moreover, the level of “structural cohesion” and “functional scattering” are measured that provide a way for assessing the structure of the experimented system. More specifically, the contributions of this work to the field of software maintenance 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 traces.
- 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 specific features as well as common features of software systems.
- Provided a measure of scattering the feature functionality over the software structure and a measure of module cohesion.
- Visualized the distribution of functions over specific features using concept lattice analysis.
- Implemented a publicly available Eclipse plug-in toolkit (Dynamic Alborz) for dynamic analysis [5].

The proposed dynamic analysis in this paper has been the foundation for our current research on the hybrid static and dynamic approaches, through: embedding run-time profiling information into a pattern-based architecture recovery technique
to control component interactions [28]; and a multi-view architecture recovery where
the structure view is reconstructed using modules and interconnections that are
resulted by growing the core functions related to the specific feature identification
in source code [29]. Currently, we are pursuing the integration of dynamic analysis
with a two-phase design pattern recovery technique, and we intend to apply the
proposed execution pattern mining towards identifying interaction patterns among
web-based distributed systems.

References
URL = http://www.cas.mcmaster.ca/~sartipi/Alborz/dynamic/.
the Eleventh International Conference on Data Engineering, pages 3–14, Washington,
European software engineering conference held jointly with the 7th ACM SIGSOFT
international symposium on Foundations of software engineering, pages 216–234, Lon-
automated performance diagnosis (distinguished paper). Lecture Notes in Computer
[12] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in dis-
tributed systems. Technical report, Software Engineering Research Center (SERC),
2004.
means of concept analysis. Fifth European Conference on Software Maintenance and
system-user interaction traces. In SEKE '02: Proceedings of the 14th international
conference on Software engineering and knowledge engineering, pages 447–454, New
[16] M. Ernst. Static and dynamic analysis: synergy and duality. In ICSE Workshop on
[18] O. Greevy and S. Ducasse. Correlating features and code using a compact two-


[38] A. Zaidman, T. Calders, S. Deweyer, and J. Paredaens. Applying webmining tech-
