# Application of Execution Pattern Mining and Concept Lattice Analysis on Software Structure Evaluation

Kamran Sartipi and Hossein Safyallah
Dept. Computing and Software, McMaster University
Hamilton, ON, L8S 4K1, Canada
{*sartipi, safyalh*}*@mcmaster.ca*

## Abstract

*Software maintenance activities for producing a feature-rich system tend to impair the software's structure into an unshaped and cost-prone legacy system. Thus, it is desirable to keep track and measure the impacts of the newly added features on the structure of the software system. The proposed technique in this paper is based on extracting frequent patterns in the execution traces of a software system using a pattern discovery technique. The patterns represent functionalities that correspond to the feature specific scenarios. In a further step, the generated execution patterns are distributed on a concept lattice to separate feature specific patterns from commonly used patterns. The proposed technique allows for assigning software features onto the software system modules and provides a means for assessing the degree of functionality scattering among the system modules. Consequently, we measure the impact of individual features on the structure of the system. A case study on the Unix Xfig drawing tool is used to present the accuracy of the approach.*

KEYWORDS: Software Maintenance; Dynamic Analysis; Execution Pattern Mining; Concept Lattice; Feature.

## 1. Introduction

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 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 impair on the structure of the software system and assess the current state of the resulting legacy system. In this paper, we propose a novel approach to assess the structural merit of the software system based on the degree of functional scattering of software features among the structural modules. In the proposed approach, 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.

The proposed approach takes advantage of dynamic analysis, data mining technique sequential pattern discovery, and concept lattice analysis to provide comprehensive information about the software system from different aspects. A key characteristic of this approach is to identify a mapping between the implementation of the software features and the structure of the system to assess the impact of a feature on the structure of the system. We execute a set of carefully defined scenarios that are specific to certain software features in order to reveal the realization of the scenario functionality within the software system modules. In this paper, we propose a novel approach to dynamic analysis and structural evaluation of a software system that contributes to the software maintenance field by the followings: i) identification of the set of core functions that implement both specific features and commonly used features of a software system; ii) providing a measure of scattering of the feature functionality to the structural modules as well as a measure of cohesion for a structural module; and iii) visualizing the functional distribution of specific features on a lattice using concept lattice analysis. The approach uses techniques such as sequential pattern discovery from data mining domain, loop-based redundant trace elimination from string processing field, as well as the visualization power of the concept lattice analysis, to automatically extract the functionality of the specific features as well as the common features of a software system. 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" [11]) than using static structural techniques such as inter-/intra-edge connectivity of the components.

The remaining of this paper is organized as follows: Section 2 describes the related work. In Section 3 the proposed framework for dynamic analysis is presented. Section 4 outlines the data cleansing process. Section 5 presents an overview of the techniques used in our approach. Section 6 describes the execution pattern analysis and structural evaluation in more detail. In Section 7 Xfig system is analyzed as the case study. Finally, the paper concludes in Section 8.

## 2. Related Work

In this section, we briefly present the closest related approaches and compare them with our work.

The applications of concept lattice analysis in software engineering have been studied for both static and dynamic analyzes. In static analysis, C. Lindig et al. [10] and Siff et al. [12] have applied concept lattice analysis in order to modularize legacy software systems; where the relationships between procedures and global variables in legacy code are investigated to extract and evaluate the candidate modules. In contrast to the above concept lattice analyzes, our approach exploits dynamic behavior of a software system to evaluate its structural properties. In dynamic analysis approaches using concept lattice, Eisenbarth et al. [6, 7] introduced a formal lattice to locate computational units that implement a certain feature of the software system. Similarly, we use the relation between scenarios and computational units (i.e., functions, methods, procedures) that are invoked during the scenario execution. However, our technique reduces the number of invoked computational units in each scenario execution to those reside in the execution patterns in order to relax the complexity of the resulting lattice.

In a different context, El-Ramly et al. [8] apply sequential pattern mining on user-interaction traces of the system, in order to reveal interaction patterns between graphical user interface components.

The following approaches use program slicing techniques to support software maintenance activities. Gallagher et al. [9] and Beszedes [3] apply (static and dynamic) program slicing in order to isolate the effects of a change in the software source code. The method allows maintainers to identify the statements and variables that may require modifications as a result of this change.

Overall, in the proposed technique we amalgam the advantages of data mining techniques with mathematical concept lattice to explore non-trivial feature-based execution patterns; consequently we identify individual modules that implement software features as a means to measure the module's structural merit.

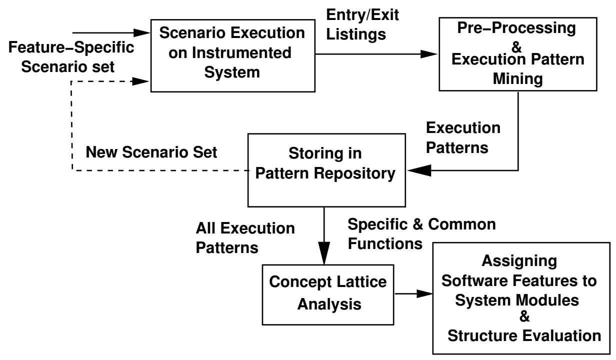
Feature–Specific Scenario set
Scenario Execution on Instrumented System
Entry/Exit Listings
Pre–Processing & Execution Pattern Mining
Execution Patterns
New Scenario Set
Storing in Pattern Repository
All Execution Patterns
Specific & Common Functions
Concept Lattice Analysis
Assigning Software Features to System Modules & Structure Evaluation

**Figure 1. Proposed framework for assigning software feature onto the system modules.**

## 3. Proposed Framework

Figure 1 illustrates different steps of the proposed framework for assigning software's feature-functionality onto the system modules. The framework takes advantage of the relation discovery power of *data mining* and *concept lattice analysis* and provides a means for measuring the impact of individual features on the structure of the system. The operations in the framework of Figure 1 are as follows.

**STEP 1: extracting execution traces**. 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 share 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 [1] to generate text messages at the entrance and exit of each function execution. By running the group of feature-specific scenarios on the instrumented software system a set of unprocessed *entry/exit listings* of function invocations are generated. In order to make the large size of the generated traces manageable, in a *preprocessing* 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 trimmed execution traces are then fed into the execution pattern mining engine in the next stage. The preprocessing operation will be discussed

[1] Instrumentation refers to inserting particular pieces of code into the software system (source code or binary image) to observe its runtime behavior.

in more details in Section 4.

**STEP 2: execution pattern discovery**. In this step, 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 operation will be discussed in more details in Section 5.1. 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 execution patterns corresponding to specific features within a group of scenario sets. This is performed by identifying those execution patterns that are specific to a single software feature within one scenario set (namely *feature-specific patterns*). Similarly, we identify the execution patterns that are common among all sets of scenarios (namely *common patterns*). Each execution pattern has significance in localizing an important feature of the subject software system. However, 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. 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 patterns and common patterns. Finally, by associating the functions of the generated clusters to the system's structural modules, i.e., files of the system, and applying two metrics for measuring *module cohesion* and *feature functional scattering*, we measure the impact of individual features on the structure of the software system. This operation is discussed in Section 6.

## 4 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 dynamic analysis. Therefore, before using the extracted entry/exit listing in further stages, redundancies in a trace that are produced by program loops and recursive function calls should be eliminated. For the analysis in this paper, we ignore recursive function traces and focus on pruning the loop-based redundancies.

In doing this, we transform the *entry/exit listing* (discussed in section 3) that is generated by instrumenting the software system (using *Aprobe* tool [13]) 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.

In this context, the loop redundancy removal problem is reduced to identification of identical subtrees that are repeated under a particular node. In order to find the repetitions that exist among subtrees of a given node, we first label these subtrees with unique IDs, where identical subtrees possess identical IDs. We then generate a string from IDs of these sibling subtrees. By applying a repetitive string finder algorithm (*Crochemore* [5]) we represent the original string (with repetitions) in the form of a new string with no repetitions. In this new string, each string repetition is shown as one instance of the repeated items that is labeled with the number of the repetitions. For example, in Figure 2(a) the string *F1,F2,F1,F2, ..., F1, F2* is transformed into a string with no repetition *(F1,F2)*$^n$ in Figure 2(b).

As a result, we keep only a subtree (among similar 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.

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

$$\ldots, Foo, F1, F1, F2, F1, F2, \ldots, F1, F2, \ldots$$

(a)

$$\ldots, Foo, F1, (F1, F2)^n, \ldots$$

(b)

**Figure 2. (a) A trace generated from Procedure Foo. (b) Loop free representation of (a).**

In Procedure Foo a piece of code that produces a long trace with repetitions of "*F1, F2*" is shown. Figures 2(a) and 2(b) represent the parts of execution trace that is produced by Procedure Foo, and the result of applying Crochemore algorithm, respectively.

## 5. Techniques for exploring patterns

In the following subsections, we briefly present the application of sequential pattern mining and mathematical concept lattice analysis. The former is used to extract highly repeated execution patterns and the later is applied on the extracted execution patterns in order to cluster the functions that exist within common / feature-specific patterns.

### 5.1. Execution Pattern Mining

A major characteristic of the run time analysis of a software system is generating execution traces with large sizes that make the task of analysis a daunting one. The effective function-trace of an intended scenario is cluttered by a large number of function calls from various places such as initialization / termination operations, utilities, and loop-based repetitions. In the rest of this subsection, 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 [2]. In this context, the sequence of all transactions corresponding to a certain customer (already ordered by increasing transaction-time) is known as a *customer-sequence*. A customer-sequence *supports* a sequence $s$ if $s$ is a subsequence 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 (namely $MinSupport$ of this pattern).

In this paper, we use a modified version of the sequential pattern mining algorithm by Agrawal [2], where an *execution pattern* is defined as a contiguous part of an execution trace (as a customer-sequence defined above) that is supported by $MinSupport$ number of execution traces. 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 particularly complicates the dynamic analysis of a software system. In the presented approach, each extracted sequential pattern consists of solely a contiguous part of different execution traces. This strategy produces meaningful execution patterns that correspond to core functions implementing specific functionalities of the system. Whereas, 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.

### 5.2. Concept Lattice Analysis

The *mathematical concept analysis* was first introduced by Birkhoff in 1940 [4]. 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 significant insight into the structure of a relation between objects and attribute-values such that each node of the lattice represents a concept.

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. The strategy to identify groups of shared attributes will be described in the next section.

## 6. Assigning Feature to Structure

In this section, we present the assignment of *software feature families* onto the software system modules, as a means of assessing the merit of the software structure. As it was discussed in the proposed framework in Section 3, we define and execute a number of feature-specific scenario sets where each scenario set targets a different software feature. Furthermore, we generate execution patterns by applying the sequential pattern mining algorithm on the loop-free execution traces. The generated execution patterns and their functions are then mapped onto a concept lattice in order to identify clusters of functions, where each cluster corresponds to a family of related software features. In the remaining of this section, we describe pattern analysis aspects and software structure assessment of the proposed approach.

**Scenario Model**

In the context of this paper, we define a scenario as a sequence of relevant features of a software system, where:

- *feature* $\phi$ is a unit of software requirements that describes a single functionality in the software system under study. $\Phi$ is the set of all available features.

- *feature family* $\Phi_\phi$ is a set of semantically relevant features to specific feature $\phi$ in the subject software system that are defined towards similar functionalities.

- *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* $\mathcal{S}_\phi$ is a set of scenario $s$ that uses specific feature $\phi$; thus $\mathcal{S}_\phi = \{s \mid s \in \mathcal{S} \wedge \phi \in \textit{setOf}^2(s)\}$.

[2]setOf(s) denotes to the set representation of sequence s

An execution pattern is treated as a sequence of functions that implement common feature(s) within a scenario set. In the following, the different kinds of execution patterns that may exist in the execution of a group of feature-specific scenario sets along with the corresponding extraction mechanisms are presented.

**Feature-specific execution pattern**
A feature-specific execution pattern corresponds to the core functions that implement a targeted feature $\phi$ of a scenario set $\mathcal{S}_\phi$ (e.g., drawing a specific figure such as rectangle). Such a pattern exists in the majority of a feature-specific scenario-set $\mathcal{S}_\phi$. 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 $\mathcal{S}_\phi$.

**Omnipresent execution pattern**
An omnipresent execution 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 trace of every scenario-set $\mathcal{S}_\phi$. In order to extract such a pattern, we should use a filtering mechanism (concept lattice in section 5.2) to filter out the feature-specific patterns from this group of patterns.

**Concept Lattice Analysis**
We employ a strategy to spotlight on 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. A key property of the functions in both kinds of clusters is that these functions belong to shared *contiguous sequences* of function calls not to the shared scattered function calls within scenario set executions. In our setting for concept lattice analysis, an object is a targeted feature $\phi \in \Phi$ of a feature-specific scenario set $\mathcal{S}_\phi$, and an attribute is a function $f$ that participates in the execution patterns within $\mathcal{S}_\phi$.

In this context, the omnipresent function clusters appear in the upper region of the lattice, and feature-specific function clusters are collected by nodes in the lower region of the lattice. In the following, we define the group of concepts that are relevant to feature-specific clusters.

- *feature-specific concept* $c_\phi$ is a concept whose support consists of a single feature $\phi$. We define $F'_\phi$ to be the set of functions corresponding to $c_\phi$ on the lattice.

- we define $F_{\Phi_\phi}$ to be the set of functions that implement the feature family $\Phi_\phi$. Thus:

$$F_{\Phi_\phi} = \bigcup_{\varphi \in \Phi_\phi} F'_\varphi$$

  where $F_{\Phi_\phi}$ represents a software system *logical module* that implement the core functionality of a feature family $\Phi_\phi$.

## 6.1. Structural Cohesion and Functional Scattering

In this section, we assess the degree of distribution of collected functions of logical module $F_{\Phi_\phi}$ over the structure of the system as a means for evaluating the feature functional scattering among software modules. Moreover, we assess the degree of concentration of logical module functions $F_{\Phi_\phi}$ within a specific software module (e.g., file) to evaluate the software module's cohesion with respect to feature family $\Phi_\phi$. These two measures provide a mechanism to evaluate the impact of individual features on the structure of the software system.

In the following, $SC_{\Phi_\phi}(m)$ denotes structural cohesion of module $m$ with respect to logical module $F_{\Phi_\phi}$ and $FS(\Phi_\phi)$ denotes functional scattering of feature family $\Phi_\phi$:

- Let $M_{\Phi_\phi} = \{m_1, m_2, \ldots, m_k\}$ be the set of modules where all the functions in $F_{\Phi_\phi}$ are defined in elements of $M_{\Phi_\phi}$.

- Let $F_m$ denote the set of functions that are defined in module $m$.

- Structural Cohesion $SC_{\Phi_\phi}(m)$ of module $m$ with respect to logical module $F_{\Phi_\phi}$ is defined as:

$$SC_{\Phi_\phi}(m) = \frac{|F_m \cap F_{\Phi_\phi}|}{|F_m|}$$

- Functional Scattering $FS(\Phi_\phi)$ of feature family $\Phi_\phi$ is defined based on the distribution of functions in $F_{\Phi_\phi}$ over modules in $M$ as:

$$FS(\Phi_\phi) = 1 - \frac{\sum_{m \in M_{\Phi_\phi}} SC_{\Phi_\phi}(m)}{|M_{\Phi_\phi}|}$$

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

## 7. Case Study

In this section, we discuss the result of applying the proposed dynamic analysis technique on Xfig 3.2.3d [1]. Xfig 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) through operations such as copy, move, edit, scale, and rotate. In order to extract the core functions that implement a specific feature (e.g., Circle-Diameter in Table 2) we define a group of feature-specific scenarios to target this feature and execute on the instrumented Xfig system to obtain the corresponding execution traces. After pruning the loop-based function calls (section 4) we apply the execution pattern mining process to obtain the existing execution patterns. We repeat the above process for each member of a feature-family in order to collect the corresponding execution patterns. Table 2 presents the statistical information for two feature-families of Xfig.

In a further step, we supply the resulting execution patterns to a concept lattice generation tool. 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. 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 feature families under study. The results of this evaluation are presented in Table 3.

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 1 demonstrates the results of experimentation with Xfig tool to reveal the core functions for two Xfig feature families. We focus on drawing a figure in the ellipse family (including circle and ellipse) such that each figure can be drawn in two different ways, i.e., by-radius and by-diameter. Furthermore, we expand our experiments on editing operation of the Xfig tool (i.e., copy graphical objects). The extracted logical modules are shown in Table 1 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 2 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).

| **Feature Family** | **Extracted Core Functions representing logical module $F_{\Phi_\phi}$** |
|---|---|
| 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 |

**Table 1. Results of feature to code assignment for 2 features of Xfig drawing tool.**

**Structural evaluation**. For each feature family $\Phi_\phi$ in Table 2 we inspect the Xfig source files that define the functions that implement the corresponding logical module $F_{\Phi_\phi}$ of that feature family. The results of measuring the structural cohesion $SC_{\Phi_\phi}(m)$ of these files are presented in Table 3. These results indicate that file d_ellipse has high cohesion with respect to logical module of feature family *Ellipse* and files e_copy, and e_move are also highly cohesive with respect to feature family *Copy*. 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 $F_{\Phi_\phi}$ 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 structural modules e_copy and e_move which indicates low scattering.

We also adopt a minimum threshold value of 10% in order to consider a structural module in the calculation of the above measurements. The results in Table 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 for any change to the *Ellipse* feature family. Similarly, changes to the *Copy* feature family would not propagate throughout the system which indicates less maintenance activities.

| Feature Family | Specific Feature of Xfig | Number of Different 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 |

**Table 2. Results of execution trace extraction and execution pattern mining for 2 Xfig feature families.**

| Feature Family $\Phi_\phi$ | Contributed File ($m$) | $\|F_m\|$ | $\|F_m \cap F_{\Phi_\phi}\|$ | Structural Cohesion $SC_{\Phi_\phi}(m)$ | Functional Scattering $FS(\Phi_\phi)$ |
|---|---|---|---|---|---|
| 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% | |

**Table 3. Structural cohesion and feature functional scattering measures for 2 Xfig feature families.**

## 8. Conclusions

In this paper, 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 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. 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 proposed technique has been applied on a medium size interactive drawing tool 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 provides a way for assessing the structure of the experimented tool.

## References

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

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In Proceedings of *ICDE '95* , pages 3–14, 1995.

[3] A. Beszedes. Union slices for program maintenance. In Proceedings of *ICSM '02*, pages 12–21, 2002.

[4] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1st edition, 1940.

[5] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Process Letters*, 12(5):244–250, 1981.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In Proceedings of *CSMR '01*, pages 176–179, 2001.

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

[8] M. El-Ramly, E. Stroulia, and P. Sorenson. Recovering software requirements from system-user interaction traces. In Proceedings of *SEKE '02:*, pages 447–454, 2002.

[9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Eng.*, 17(8):751–761, 1991.

[10] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In Proceedings of *ICSE '97*, pages 349–359, 1997.

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

[12] M. Siff and T. W. Reps. Identifying modules via concept analysis. In Proceedings of *ICSM '97*, pages 170–179, 1997.

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