# Dynamic Analysis of Software Systems using Execution Pattern Mining

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

## Abstract

*Software system analysis for extracting system functionality remains as a major problem in the reverse engineering literature and the early approaches mainly rely on static properties of software. In this paper, we propose a novel technique for dynamic analysis of software systems to identify the implementation of the software features that are specified through a number of feature-specific task scenarios. The execution of task scenarios and application of data mining algorithm sequential pattern discovery on the generated traces allow us to extract common functionality associated with the corresponding feature-specific task scenarios. The extracted patterns are used to identify the groups of core functions that implement software features. The proposed approach can be used for program comprehension and feature to source code assignment. A case study on the Unix Xfig drawing tool has been provided.*

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

## 1. Introduction

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 [7]. However, static analysis suffers from the lack of enough semantics for design or 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 the software reverse engineering [8, 3]. 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, software execution visualization, and feature to code assignment. In this work, we address the latter problem. Dynamic analysis with its suitability in extracting system functionality has several challenges compared to the static analysis: i) a static analysis usually generates a complete set of software facts 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 affects the applicability of the dynamic analysis; and iii) the large sizes of the execution traces that are caused by program constructs such as loops and recursions may disfunction the whole dynamic analysis.

In this paper, we propose a novel 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 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 spotlight on the feature-related system functionality. In this context, we obtain high-frequent patterns in execution traces, where a post-processing of the generated execution patterns will allow us to separate the more general functionalities (e.g., starting/terminating operations and common utility functions) from the specific functionality of the task scenarios.

The contribution of this paper include: managing the large sizes of the execution traces using data mining techniques that leads us to automatically identify specific and general software feature functionalities within the source code.

The remaining of this paper is organized as follows: In Section 2 the proposed framework for dynamic analysis is presented. In Section 3 we discuss the execution pattern mining analysis. Section 4 provides a case study using Xfig drawing tool. Section 5 addresses the related work; and finally, Section 6 concludes our discussion.

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE

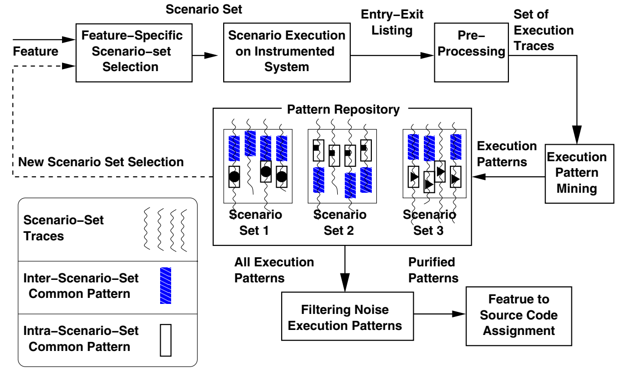
Scenario Set
Feature
Feature–Specific Scenario–set Selection
Scenario Execution on Instrumented System
Entry–Exit Listing
Pre–Processing
Set of Execution Traces
Pattern Repository
New Scenario Set Selection
Execution Patterns
Execution Pattern Mining
Scenario Set 1
Scenario Set 2
Scenario Set 3
Scenario–Set Traces
Inter–Scenario–Set Common Pattern
Intra–Scenario–Set Common Pattern
All Execution Patterns
Purified Patterns
Filtering Noise Execution Patterns
Featrue to Source Code Assignment

**Figure 1. Proposed dynamic analysis framework to identify feature functionality in the source code.**

## 2. Proposed Framework

Figure 1 illustrates an overview of the proposed framework for dynamic analysis of software systems. The framework allows us to localize the patterns of program executions that correspond to specific features of the task scenarios. This process consists of two stages: *Execution Pattern Extraction* and *Pattern Analysis*. In the remaining of this section these stages are briefly described.

**Stage 1** (*Execution Pattern Extraction*): based on the application domain, available documents, and user's familiarity with the subject 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 all scenarios that examine the *zooming* ability of the software would constitute such a feature-specific scenario set. In the next step, the software system is instrumented [1] to produce the name of each function both at the entry and exit of the function execution. Therefore, executing the feature-specific scenarios on the instrumented software system generates a set of *entry/exit listings* that are then transformed into a set of function execution traces. In a further preprocessing step, all redundant function calls caused by the cycles of the program loops are eliminated that significantly reduce the large sizes of execution traces. Finally, in this stage we generate execution patterns by applying a sequential pattern mining algorithm on the obtained execution traces. This stage will be discussed in more details in Section 3.

**Stage 2** (*Execution Pattern Analysis*): each execution pattern is a potential candidate group of functions that implement common feature(s) of a scenario set. We employ a strategy to spotlight on the execution patterns corresponding to specific features within the scenario sets. This is performed by identifying those execution patterns that are specific to a single software feature in a scenario set (namely *intra-scenario-set common patterns*). Similarly, we identify the execution patterns that are common among all sets of scenarios (namely *inter-scenario-set common patterns*). In Figure 1 a sketch of the scenario-set execution traces and *intra- / inter-* scenario-set common patterns are shown. As the last step, we identify the functions in the intra/inter-scenario-set patterns in order to locate the feature functionality in the source code. This stage is discussed in Section 3.2.

## 3. Execution Pattern Mining

In this section we describe the application of a data mining technique, namely *sequential pattern mining*, to discover a set of function sequences that implement certain system features. In the data mining literature a sequential pattern mining technique is used to extract frequently occurring patterns of purchased items within the sequences of customer transactions [2]. In this context the sequence of all transactions corresponding to a certain customer that is already sorted by increasing transaction-time, is known as a *customer-sequence*[2]. A customer-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 (namely $MinSupport$ of this pattern).

In the proposed approach, we use a modified version of the sequential pattern mining algorithm by Agrawal [2]. In our implementation an *execution pattern* is defined as a contiguous part of an execution trace that is supported by $MinSupport$ number of execution traces. This strategy produces core functions that implement specific features of the system. By extending the definition of the execution pattern to include noncontiguous function invocations, we can extract function patterns that implement more general functionality; 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 which drastically increases the time/space complexity of the dynamic analysis.

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

[2] In the context of this paper, a function execution trace represents a customer-sequence.

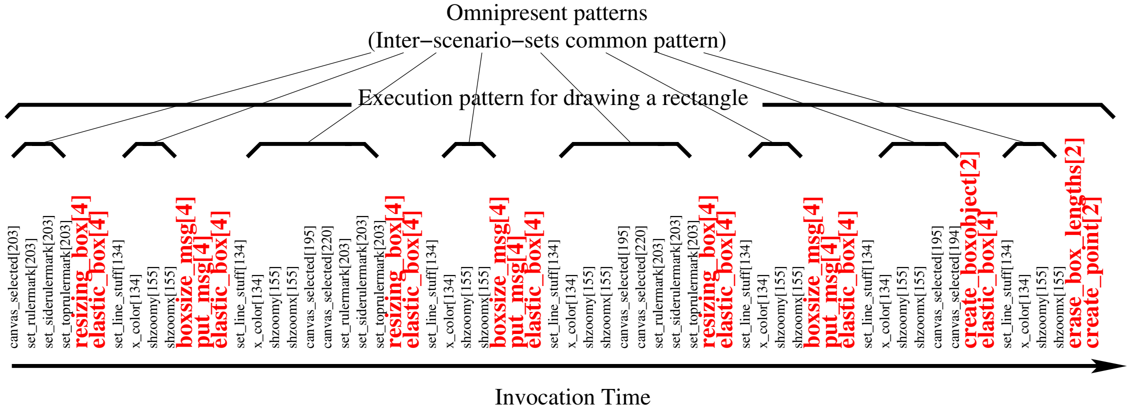

**Figure 2. A first generation pattern extracted for drawing a rectangle in Xfig with the highlighted second generation patterns along with their support counts.**

### 3.1. Categories of Execution Patterns

A large group of patterns are generated in the execution pattern mining that camouflage the important patterns and make the task of core functionality extraction a non-trivial and daunting task. A typical execution pattern can be categorized as one of the following categories. i) Patterns corresponding to the core functions that implement the targeted feature of a feature-specific scenario set; in Figure 1 this category is referred to as intra-scenario-set common patterns. ii) Omnipresent patterns that are common to almost every task scenario of the software system; in Figure 1 these patterns are referred to as inter-scenario-set common patterns. iii) Noise patterns that do not contribute to a major system functionality. The following strategies are used to extract shared patterns corresponding to a set of task scenarios.

*Strategy I*: given all execution patterns corresponding to *a single* feature-specific scenario-set (i.e., sharing a specific feature), those patterns that are generated by the majority of the scenarios most likely implement the shared feature of the scenario set. In order to implement this strategy we increase the level of $MinSupport$ for the generated execution patterns to a number that covers the majority of the scenarios in the corresponding scenario-set. In this way, the noise patterns will be removed from the extracted execution patterns. However, the resulting patterns still consist of both omnipresent and feature-specific patterns.

*Strategy II*: given all execution patterns corresponding to *a group* of feature-specific scenario-sets, each with a different specific feature, the execution patterns that are shared among the majority of the scenarios (i.e., omnipresent patterns) most likely implement the general features of the system.

In the next subsection, we describe our proposed approach to extract each type of execution patterns with regard to the above strategies.

### 3.2. Separating Execution Patterns

As discussed above, the generated execution patterns during *strategy I* include both feature-specific and omnipresent execution patterns. We apply the execution pattern mining twice in order to separate the two types of execution patterns. The steps are as follows:

- *First generation patterns*. A group of feature-specific scenario sets are defined, where each scenario set targets a different feature of the software, and the execution patterns corresponding to each of these scenario sets are extracted.

- *Second generation patterns*. For the second time, we apply the execution pattern mining on the collection of the first generation patterns. The second generation patterns with small support (e.g., less than 5%) correspond to the feature-specific patterns. However, the patterns with a large support (e.g., more than 25%) correspond to the omnipresent execution patterns.

Figure 2 depicts a part of a first generation pattern corresponding to Xfig drawing rectangle feature, where the second generation patterns are highlighted along with their support counts. The functions with bold fonts are feature-specific patterns with small support that perform significant role in specifying the boundary region for drawing a new rectangle on the screen.

| Feature | Extracted Core Functions |
|---|---|
| Draw Circle | resizing_cbr, elastic_cbr, pw_curve, create_circlebyrad center_marker, create_ellipse, add_ellipse, list_add_ellipse set_lastspline, redisplay_ellipse, ellipse_bound, draw_ellipse overlapping, debug_depth, circlebyradius_drawing_selected |
| Draw Rectangle | resizing_box, elastic_box, boxsize_msg, create_boxobject create_point, create_line, add_line, box_drawing_selected |
| Draw Spline | create_spline, make_sfactor, create_sfactor, add_spline last_spline, set_latestspline, redisplay_spline, spline_bound approx_spline_bound, draw_spline, compute_closed_spline |
| Scale | erase_objecthighlight, init_center_scale, init_scale_line scaling_line, adjust_box_pos, elastic_scalepts, fix_scale_line rescale_points, scale_arrows, scale_arrow, scale_linewidth |
| Move | init_arb_move, init_move, init_line_dragging set_action_on, elastice_moveline, elastic_links, moving_line place_line, erase_lengths, place_line_x, adjust_pos set_lastposition, set_newposition, move_selected |

**Table 1. Extracted core functions corresponding to 5 specific Xfig features.**

## 4. Case Study

In this section, we present the results 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 is used to draw and manipulate graphical objects (circle, ellipse, line, spline, rectangle, and polygon) through operations such as copy, move, delete, edit, scale, and rotate.

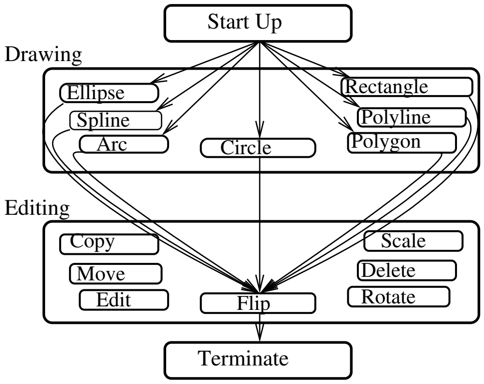

**Figure 3. A feature-specific scenario set that target the Xfig operation "Flip".**

Figure 3 depicts the adopted strategy to single out a targeted feature by the 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. In Figure 3 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 3 form a feature-specific scenario set, where the *flip* operation is the specific feature.

| Xfig Feature | Number of Different Scenarios | Average Trace Size | Number of Extracted Patterns | Average Pattern Size |
|---|---|---|---|---|
| Draw Circle | 10 | 8143 | 48 | 32 |
| Draw Rectangle | 10 | 5510 | 43 | 46 |
| Draw Spline | 10 | 17000 | 61 | 62 |
| Scale Objects | 4 | 6580 | 38 | 47 |
| Move Objects | 4 | 11887 | 31 | 53 |

**Table 2. 5 Xfig feature-specific scenario sets and their dynamic characteristics.**

We follow the steps defined in subsection 3.2 in order to extract the core functions that implement both specific features and general features of the Xfig drawing tool. Table 1 illustrates a group of Xfig features along with the extracted core functions that implement those features. In the remaining of this section, we discuss the important properties of the proposed pattern based dynamic analysis technique using the Xfig case study:

**Reducing the complexity of analysis**: Table 2 represents the attributes of a group of 5 feature-specific scenario sets that we use in the analysis process. This table illustrates an important aspect of the approach where the scope of the dynamic analysis has been significantly reduced from huge sizes of the execution traces (Average Trace Size in the range of thousands of functions per trace) to the manageable sizes of the execution patterns (Average Pattern Size with tens of functions per trace).

**Extracting non-visible features**: in addition to more visible system functionalities such as: software initialization / termination and major software features, the execution patterns uncover other less visible system functionalities, including: mouse pointer handling, canvas view updating, and side ruler management. This property is illustrated in Table 3.

**Preserving the sequence of operations**: in contrast to the static analysis of a software system or concept lattice based dynamic analysis discussed in the related work, the proposed pattern based dynamic analysis will preserve the time sequence of the function invocations in the result of the analysis. This property enables the user to get more insight into the system functionality using the control dependency between the functions in the execution patterns.

| Xfig Functionality | Extracted Core Functions |
|---|---|
| Side-Ruler Management | set_rulermark, set_siderulermark<br>set_toprulermark, null_proc |
| Canvase Updating | canvas_exposed, clear_canvas<br>canvase_selected |
| Mouse Pointer Handling | draw_mousefun_canvas, draw_mousefun<br>clear_mousefun, draw_mousefn2<br>draw_mousefun_msg, mouse_title |
| Draw Line | set_line_stuff, x_color, shzoomy, shzoomx |

**Table 3. Less visible Xfig functionalities (left) and their corresponding functions.**

## 5. Related Work

In this section, we present different approaches to dynamic analysis of a software system that relate to our work. Execution traces have been used in different reverse engineering and program understanding activities. Fischer et al. [5] used execution traces as clues for tracing the evolution of a software system. In [11], Zaidman proposed a heuristic exploration to execution traces that aims at clustering execution traces of recurring events. Hamou-Lhadj et al. [6] applied fan-in analysis to the class dependency graph to extract a high level view of the subject software system.

In a different context, El-Ramly et al. [4] applied a sequential pattern mining technique to find interaction patterns between graphical user interface components. In the work of Zaidman [10] a web-mining technique is applied on program dynamic call graphs that supports the program comprehension. Similar to our approach, the above approaches use mining techniques on the execution traces.

N. Wilde et al. [9] proposed a set difference approach to execution traces for locating software features; where the set of functions in the related scenario executions are differentiated to extract a 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. Eisenbarth et al. [3] proposed a formal concept lattice analysis to locate computational units that implement a certain feature of the software system.

## 6. Conclusions

In this paper, we proposed a novel approach to dynamic analysis of a software system as an application of sequential pattern mining on program execution traces. The resulting execution patterns extract valuable information out of noisy execution traces. The proposed approach is centered around a set of task scenarios that share specific software features. The whole process consists of steps such as: software instrumentation; feature-specific scenario set selection; loop-based sub-trace elimination; execution pattern extraction; execution patterns purification; and finally interpretation of the patterns. The proposed technique has been applied on a medium size interactive drawing tool with very promising results in extracting both feature specific and common patterns of execution traces. As a future work we will look at effective pruning methods for the execution trace generation to allow analysis of very large traces over 100K functions. (e.g. Apache, MySql).

## References

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

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

[3] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE TSE*, 29(3) 210 – 224, 2003.

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

[5] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind. System evolution tracking through execution trace analysis. In *IWPC'05*, pages 237–246, 2005.

[6] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR'05*, pages 112–121, 2005.

[7] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Recognizers for extracting architectural features from source code. In *WCRE'95*, pages 252–261, 1995.

[8] T. Richner and Stephane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM'99*, page 13, 1999.

[9] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *JSM'95*, 7(1):49–62, 1995.

[10] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR'05*, pages 134–142, 2005.

[11] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR'04*, page 329, 2004.