Application of Dynamic Analysis in Reverse Engineering

- Existing Dynamic Analysis approaches
  - Execution trace analysis: aspect mining, clustering, performance analysis, program slicing.
  - User-system interaction analysis: recovery of behavior patterns.

We use Dynamic Analysis to:
- Identify software functionality (feature) in source code
  - Traditionally, static analysis was used to locate function templates in source code.
  - We generate patterns of execution traces to identify the implementation of software features in source code by the means of task scenarios.
- Incorporate semantics into static analysis
  - Using feature-to-code assignment to find core functionality of the clustering techniques.
  - Providing metrics to evaluate structural properties of software systems.

Proposed Framework:
Dynamic Analysis using Execution Pattern Mining

Feature-Specific Scenario Set

- A scenario is the unit of the system functionality (e.g., flipping a figure).
- A task scenario defines the user-system interaction in the form of a sequence of software system features (operations) in an informal or semi-formal manner.

Feature-Specific Scenario Set:
- Start, Draw Ellipse, Flip, Exit
- Start, Draw Spline, Flip, Exit
- Start, Draw Arc, Flip, Exit
- Start, Draw Rectangle, Flip, Exit
- Start, Draw Polygon, Flip, Exit

Software Instrumentation

- Inserting particular pieces of code (probe) into the software's source code or binary image to extract dynamic information from the running system.

We instrument the software system to generate text messages at both entrance and exit of each function, namely Entry/Exit listings.
To eliminate program loops:
- Extracted entry-exit listings have lots of redundancies and repetitions caused by program loops that must be eliminated.

To eliminate program loops:
- Represent the Entry-Exit listing as a dynamic call tree:
  - Nodes represent functions.
  - Edges represent function calls.
  - Assign identical IDs to the nodes with identical sub-trees (nested calls).
- Prune the dynamic call tree by removing multiple instances of nodes with identical sub-trees from top to bottom.
- Generate the execution trace by a depth first traversal on the pruned tree.

Preprocessing (cont'd)

Preprocessing:
Example

Preprocessing:
Tree Pruning (cont'd)

Preprocessing:

Pruning is a 4 steps process to eliminate loop-based redundancies in a dynamic call tree.
1. Build a string representation of the sub-tree IDs rooted at each particular node.
2. Extract repetitions from the original string (with repetitions) using a string repetition finder algorithm, e.g., crochemore.
3. Represent the original string in the form of instances of repetitions and their corresponding number of repetitions.
4. Keep sub-trees that correspond to a single instance of each repetition.

Sequential Pattern Mining

Given:
- A group of items (e.g., coke, pen).
- A group of 'transaction sequences', where each transaction sequence belongs to a customer, and the transactions are ordered according to the transaction-time.

Applying a sequential pattern mining on this set of transaction-sequences reveals the common maximum sequences of items.

Interesting relationships among the items can be found.
For example in a computer bookstore, we may find that 10% of the customers first buy a C book then a C++ book and then a Java book.

Sequential Pattern Mining ...

Customer Id | Customer-Sequence
-------------|---------------------
1            | Java (100) Coke (30)
2            | Java, Coke, Java (20), Coke (20)
3            | Java, Coke (20), Java (20), Coke (20)
4            | Java, Coke (20), Java (20), Coke (20)
5            | Java, Coke (20), Java (20), Coke (20)

Customer-Sequence Version of the Database

Sequential Patterns

<Java> (100)
<Java> Coke (30)
<Java> Coke (20)
<Java> Coke (20)
<Java> Coke (20)

The Pattern Set (minimum support is 2)

Sequential Pattern Mining ...

Feature 1 | Feature 2 | Feature 3
----------|----------|----------
1          | F4, F5, F6, F7, F8 | F4, F5, F6, F7, F8, F9
2          | (Java, Coke)       | (Java, Coke)       |
3          | F10, F11, F12      | F10, F11, F12      |
4          | (Java, Coke)       | (Java, Coke)       |
5          | F20, F21, F22      | F20, F21, F22      |
6          | F10, F11, F12      | F10, F11, F12      |
7          | F10, F11, F12      | F10, F11, F12      |
8          | F10, F11, F12      | F10, F11, F12      |
9          | F10, F11, F12      | F10, F11, F12      |
10         | F10, F11, F12      | F10, F11, F12      |

Feature 1   | Feature 2   | Feature 3   | Common pattern
-------------|-------------|-------------|-----------------
F10         | F11         | F12         | Noise pattern
F13         | F14         | F15         | Feature-specific
Proposed Framework for Dynamic Analysis using Execution Pattern Mining

Pattern Generation
- Feature-specific patterns: core functions that implement the specific feature of a scenario-set.
- Common patterns: sequences of functions that appear in almost every executed scenario (e.g., system initialization and termination, mouse movement, drawing commands).

Second sequential pattern mining is performed to separate two categories of patterns:
- Step 1: first sequential pattern mining with high min-support extracts both feature-specific and common patterns.
- Step 2: second sequential pattern mining on the collection of all results of “Step 1” separates two patterns categories.
  - Patterns with small supports (e.g., less than %10) are feature-specific.
  - Patterns with large supports (e.g., more than %80) are common.

Identifying Features in Source Code … (Second Pattern Generation)

Case Study: Xfig Drawing Tool

The results of execution pattern mining for a collection of 3 different Xfig features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specific Number of Different Operations</th>
<th>Average Process Time</th>
<th>Average Function Time</th>
<th>Number of Extracted Patterns</th>
<th>Average Function Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ellipse</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>circle</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>line</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Characteristics of the proposed technique:
- Pre-processing (loop elimination) drastically reduces the sizes of the execution traces.
- Post-processing (second pattern or concept lattice) reduces the overwhelming number of execution patterns that are generated.

Concept Lattice Analysis

- Concept Lattice is generated from Context Table
- Each lattice node is a concept that may have objects and attributes.
- Every object has all the attributes that appear in that node or all nodes above it.
- Each attribute belongs to all objects that are in that node or every node below that node in the lattice.

Lattice represents the structure of the relations among entities in a database.

- Concept Lattice is generated from Context Table
- Each lattice node is a concept that may have objects and attributes.
- Every object has all the attributes that appear in that node or all nodes above it.
- Each attribute belongs to all objects that are in that node or every node below that node in the lattice.

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

context table concept lattice

Experiment with Xfig Drawing Tool …

Extracted core functions for Xfig features.
Less visible common Xfig features and their functions

<table>
<thead>
<tr>
<th>Xfig Functionality</th>
<th>Extracted Case Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Side-Order</td>
<td>scCalendar, scCalendarxmark</td>
</tr>
<tr>
<td>Management</td>
<td>scCalendar, scDeploymentmark, scCalendarxmark, scDeploymentxmark, scGantt, scGantt2</td>
</tr>
<tr>
<td>Curve</td>
<td>curve, curve2, curve3, curve4, curve5</td>
</tr>
<tr>
<td>Updating</td>
<td>curve, curve2, curve3, curve4, curve5</td>
</tr>
<tr>
<td>Mouse</td>
<td>draw_polyline, draw, move, moveto, fill, fillshape</td>
</tr>
<tr>
<td>Rastering</td>
<td>draw_polyline, draw, move, moveto, fill, fillshape</td>
</tr>
<tr>
<td>Draw Line</td>
<td>scLine, scLine2, scLine3, scLine4, scLine5</td>
</tr>
</tbody>
</table>

Case Study: Xfig

The Execution Trace for scenario “Drawing and Flipping Rectangle” is annotated with Descriptions of Execution Patterns.

We proposed:
- A pattern-based approach to dynamic analysis of software systems that employs data mining techniques to extract functional information out of noisy execution traces.
- A measure of functionality scattering of a feature among structural modules as well as a measure of cohesion for each structural module.
- A method of visualizing the functional distribution of specific features on a lattice using concept lattice analysis.
- The technique deals with scalability, through:
  - Reducing the size of execution traces by eliminating the loop-based repetitions.
  - Replacing large sizes of the loop-free traces using data mining techniques.
- A method for assigning semantics to the static analysis of a software

References

Pattern Analysis:

Concept Lattice Analysis (cont’d)

- An object is a targeted feature of a feature-specific scenario set $\mathcal{S}_\phi$.
- An attribute is a function that participates in the execution patterns within $\mathcal{S}_\phi$.
- A feature-specific concept $\mathcal{C}_\phi$ is a concept with a single object (feature) $\phi$. We define $\mathcal{C}_\phi$ to be the set of functions that appear on $\mathcal{C}_\phi$.
- A logical module $\mathcal{M}_\phi$ is the set of functions that implement feature family $\Phi_\phi$.

$$\mathcal{M}_\phi = \bigcup_{\Phi \in \mathcal{S}_\phi} \mathcal{F}_\phi$$

Structural Evaluation

- Let $\mathcal{M}_\phi = \{m_1, m_2, \ldots, m_n\}$ be the set of modules where all the functions in $\mathcal{M}_\phi$ are defined in elements of $\mathcal{M}_\phi$.
- Let $\mathcal{F}_\phi$ to the set of functions that are defined in modules $m_i$.

Structural cohesion of module $m_i$ with respect to feature family $\Phi_\phi$

- $\mathcal{C}_\Phi(m_i) = \frac{|\mathcal{F}_\phi \cap \mathcal{F}_{\Phi_i}|}{|\mathcal{F}_{\Phi_i}|}$

 Functional scattering of feature family $\Phi_\phi$

- $\mathcal{F}(\Phi_\phi) = 1 - \frac{\sum_{m_i \in \mathcal{M}_\phi} \mathcal{C}_\Phi(m_i)}{|\mathcal{M}_\phi|}$

Formal Definitions:

Scenario, Feature

- A software feature (of type text) is a unit of software requirements that describes a single system functionality.
- A scenario is modeled as a sequence of features, i.e., $\mathcal{S} = [\phi_1, \phi_2, \ldots, \phi_n]$.
- A feature-specific scenario set $\mathcal{S}_{\Phi_i}$ is a set of scenarios that all share a specific feature: $\mathcal{S}_{\Phi_i} = \{s \in \mathcal{S} \mid \exists \mathcal{F} \in \mathcal{S}_\phi \}$. where $\mathcal{S}$ is the set of all system scenarios.
- A feature family $\Phi_\phi$ is a set of semantically relevant features to specific feature $\phi$.

Functional Pattern Mining

- Let $\mathcal{F}$ be the set of all function names in the subject software system.
- An execution trace $T$ is a sequence of function names from $\mathcal{F}$.
- Let Repository $\mathcal{R}_{\mathcal{S}_\phi}$ be the set of all extracted traces according to the execution of task scenarios in feature-specific scenario set $\mathcal{S}_{\Phi_i}$.
- An execution pattern $P \in T$ is defined as a contiguous sequence of functions from $\mathcal{F}$ that is supported by at least $\text{MinSupport}$ number of the execution traces in the repository $\mathcal{R}_{\mathcal{S}_\phi}$.

An execution trace $\mathcal{T}$ supports execution pattern $P$ if $P$ is a subsequence of $\mathcal{T}$.

Each execution pattern extracts the sequence of functions that implement a common functionality within each feature-specific scenario set $\mathcal{S}_{\Phi_i}$.

Formal Definitions:

Execution Pattern Mining

- Provides lattice representation for the binary relation $\mathcal{R}$ between objects $\mathcal{O}$ and their attribute-values $\mathcal{A}$.
- Provides a means for clustering objects based on their common attributes.
- Provides a separation method for attributes based on their sharing level.
Concept Lattice Analysis
(cont’d)

- In the binary relation \( R \) between Objects \( O \) and attributes \( A \):
  - The triple \( C = (O, A, R) \) is called a **formal context**.
  - For any set of objects \( O \subseteq O \), we define \( shared_A(O) \) as the set of shared attributes among objects in \( O \).
  - For any set of attributes \( A \subseteq A \), we define \( shared_O(A) \) as the set of objects whose sharing all attributes in \( A \).
  - Concept \( C \) is defined as the pair \( c = (O, A) \) such that:
    \[ O = shared_O(A) \land A = shared_A(O) \]