Plan

1. Introduction & problem statement
2. State of the art
3. Proposed approach
4. Conclusion & Perspectives
1. Introduction & problem statement

- Software evolution is inescapable.
- For developers, it is crucial to understand the structure of the system before attempting to modify it.
- The global understanding can not be achieved by just reviewing the source code.
- Instead, the architecture of the system is needed.
- **Problem 01:** "software architecture erosion"
  - Architecture descriptions are, if available, outdated and incorrect.
- **Solution:** "Control software architecture erosion"
  - Minimize erosion.
  - Prevent erosion.
  - Repair erosion: **Architecture recovery.**
Problem 02: Unreadable architectures in the case of large systems

Solution:

Mitigate the complexity of the architecture by

1. **Refinement:** with lifespans and probability of existence at run-time.
2. **Abstraction:** identification of composite structures and use of thresholds to make a visualization with a Level of Detail (LoD).

Plan

1. Introduction & problem statement
2. State of the art
3. Proposed approach
4. Conclusion & Perspectives
State of the art (1/5)

- The reconstruction process begins with a phase of system’s information extraction.
- Extracted information can be categorized as either static or dynamic.
- Several approaches were proposed
- Focus in our approach: Works that reconstruct the run-time architecture.

State of the art (2/5)


- Purpose: document an object oriented software system.
- Approach: reverse engineering of class, object, interaction, state and package diagrams.
  - The basic program representation for the static analysis to extract all diagrams is an Object Flow Graph (OFG).

1. Object Flow Graph:
   - An OFG allows tracking the lifetime of the objects from their creation along their assignment to program variables.
   - Steps for constructing an OFG:
     1. Simplify Java language into an abstract language (Declarations* Statements*).
     2. Construct the graph (nodes = program locations, edges = assignments).
public class BinaryTreeNode {
    BinaryTreeNode left, right;
    public void addLeft(BinaryTreeNode n) {
        left = n;
    }
    public void addRight(BinaryTreeNode n) {
        right = n;
    }
}

public class BinaryTree {
    BinaryTreeNode root;
    public void build() {
        root = new BinaryTreeNode();
        BinaryTreeNode curNode = root;
        While(…)
        {
            curNode.addLeft(new BinaryTreeNode());
            curNode.addRight(new BinaryTreeNode());
        }
    }
    static public void main() {
        BinaryTree bt = new BinaryTree();
        bt.build();
    }
}

Example (1/3)

Example (2/3)

• Abstract language

1- Declarations

BinaryTreeNode.left
BinaryTreeNode.right

BinaryTreeNode.addLeft(BinaryTreeNode.addLeft.n)
BinaryTreeNode.addRight(BinaryTreeNode.addRight.n)

BinaryTreeNode.root
BinaryTree.build()
BinaryTree.main()

2- Statements

BinaryTreeNode.left = BinaryTreeNode.addLeft.n
BinaryTreeNode.right = BinaryTreeNode.addRight.n

BinaryTree.root = BinaryTreeNode.BinaryTreeNode.this
BinaryTree.build.curNode = BinaryTree.root

BinaryTreeNode.addLeft.n = BinaryTreeNode.BinaryTreeNode.this
BinaryTree.build.addLeft.this = BinaryTree.build.curNode

BinaryTreeNode.addRight.n = BinaryTreeNode.BinaryTreeNode.this
BinaryTree.Node.addRight.this = BinaryTree.build.curNode

BinaryTree.main.bt = BinaryTree.BinaryTree.this
BinaryTree.build.this = BinaryTree.main.bt
2. Static object diagram

- Represents the set of objects created by a given program and the relationships holding among them.
- Objects are identified by allocation sites with a control-flow insensitive way.
- The OFG is used to determine the inter-object relationships.
- The presence of a relationship between two objects in the graph means that there is a path in the program along which the first object can reference the second through an attribute.
- The relations between the objects in this graph are a conservative super-set of those that really exist in execution time.

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Refinement</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lifespans</td>
<td>Prob. of exist</td>
</tr>
<tr>
<td>Static</td>
<td>🙁</td>
<td>🙁</td>
</tr>
</tbody>
</table>

- **Purpose**: aid understanding and reasoning about object-oriented systems
- **Approach**: AARDVARK tool for the reconstruction of object models
  
  - **Steps**:  
    1) Execute an instrumented target program that records a log of all object allocations and field writes.  
    2) Use the logs to reconstruct a snapshot of the heap for each execution.  
    3) Apply a sequence of abstraction-based operations to each heap.  
    4) Combine the results into a single object model.

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Refinement</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lifespans</td>
<td>Prob. of exist</td>
</tr>
<tr>
<td>Dynamic</td>
<td>😞</td>
<td>😞</td>
</tr>
</tbody>
</table>

- **Purpose**: allow detailed understanding by achieving hierarchy in static object diagrams.

- **Approach**: SCHOLIA technique to statically extract a hierarchical runtime architecture from object oriented code using annotations.

  - **Steps**:

    1) The developer pick an object as a starting point then use local modular ownership annotations to impose an hierarchy on objects.

    2) A static analysis extracts a global object graph from the annotated program.

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Refinement</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lifespans</td>
<td>Prob. of exist</td>
</tr>
<tr>
<td>Static</td>
<td>😞</td>
<td>😞</td>
</tr>
</tbody>
</table>

**Plan**

1. Introduction & problem statement

2. State of the art

3. Proposed approach

4. Conclusion & Perspectives
Proposed approach (1/4)

- Static information is valuable to understand the structure of the system.
  - **Limit**: extracted information is potentially true.
- Dynamic information exposes the system's behavior.
  - **Limit**: provide a partial picture of the system.
- **Proposal**
  - Refining the static architecture using the informations contained in the dynamic architecture.

Proposed approach (2/4)
Proposed approach (3/4)

- **1st Step**: Static analysis
  - Create a preliminary object graph: nodes represent objects and edges represent dependencies between objects.

- **2nd Step**: Source code instrumentation to generate execution traces
  - Targets: object creation/destruction and dependency establishment/breaking.
  - The execution of the instrumented code generates traces.
  - Each trace contains informations about:
    - Creation/destruction of objects.
    - Establishment/breaking of dependencies.
    - Timestamps.

Proposed approach (4/4)

- **3rd Step**: Refining the object graph by analyzing the execution traces
  - Inserting two kinds of labels:
    - Lifespans: measured using the creation and destruction timestamps.
    - Probabilities: the ratio of the number of times that a node/edge exist in the execution traces to the total number of execution traces.

- **4th step**: Construction of the composite structure

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Refinement</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lifespans</td>
<td>Prob. of exist</td>
</tr>
<tr>
<td>Hybrid</td>
<td>😊</td>
<td>😊</td>
</tr>
</tbody>
</table>
4. Conclusion & Perspectives

- The proposed approach allows:
  - Building a graph representing objects and their dependencies.
  - Enriching the graph with probabilities and lifespans.
  - Identifying composition relations between objects and use of thresholds.

- Perspectives:
  - Implementation of the approach.
  - Experiment the approach on a large and extensively used systems.
Thank you

Questions ?
Comments ?