Accurate Polymorphism Detection

Nevena Milojković
Software Composition Group
University of Bern
Problem

```java
public static void main(String[] args){
    AttributeFigure figure = new ComponentFigure();
    figure.basicDisplayBox(point1, point2);
}
```
public static void main(String[] args) {
    AttributeFigure figure = FigureFactory.getFigure();
    figure.basicDisplayBox(point1, point2);
}
We know this information at run-time.
Problem: Program comprehension in the presence of polymorphism

Goal: Create an accurate call-graph at code-reading-time

Idea: Compare dynamically collected results with static algorithms
# Static algorithms

<table>
<thead>
<tr>
<th></th>
<th>Class hierarchy</th>
<th>Instances per project</th>
<th>Instances per class</th>
<th>Instances per field</th>
<th>Instances per method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. UN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2. CHA</strong></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3. RTA</strong></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4. CTA</strong></td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td><strong>5. MTA</strong></td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>6. FTA</strong></td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>7. XTA</strong></td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
What is really happening?

I Don’t Know.
Collect information from a running system

Collect information about all method invocations from the project in question

Store information in a RTI (run-time information) database

Compare dynamically collected results from RTI database with static algorithms
Using Javassist to get the information

```java
public void basicDisplayBox(Point origin, Point corner) {
    bounds = new Rectangle(origin);
    bounds.add(corner);
}

public void basicDisplayBox(Point origin, Point corner) {
    Profiler.log($0, $sig, $args);
    bounds = new Rectangle(origin);
    bounds.add(corner);
}
```
figure.basicDisplayBox(origin, corner);

org.jhotdraw.contrib.ComponentFigure.basicDisplayBox(Point,Point);
org.jhotdraw.contrib.PolygonFigure.basicDisplayBox(Point,Point);
org.jhotdraw.contrib.TextAreaFigure.basicDisplayBox(Point,Point);
org.jhotdraw.figures.EllipseFigure.basicDisplayBox(Point,Point);
org.jhotdraw.figures.ImageFigure.basicDisplayBox(Point,Point);
org.jhotdraw.figures.RectangleFigure.basicDisplayBox(Point,Point);
org.jhotdraw.figures.RoundRectangleFigure.basicDisplayBox(Point,Point);
How confident are we in our results?

62% of used fields
26% of used methods
64% of used constructors
65% of used classes
• Implement more static algorithms
• Implement three-stage analysis
• Improve performance for dynamic analysis
• Run analysis on more projects
• Integrate a tool into IDE
Additional uses of the RTI database

- Usage of fields
- All methods invocations
- Study null pointer propagation
Summary

• Call graph helps source code comprehension
• Polymorphism introduces ambiguity in the call-graph
• Static algorithms give false positives
• Dynamic analysis give false negatives
• Their combination could yield more accurate results, at a reasonable cost, to support the developer