Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study

Davy Landman*, Alexander Serebrenik†, Jurgen J. Vinju†
* Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Davy.Landman, Jurgen.Vinju}@cwi.nl
† Eindhoven University of Technology, Eindhoven, The Netherlands
{a.serebrenik,j.j.vinju}@tue.nl

This work has been supported by the NWO TOPCO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”
Our initial context:
Is “reflection” going to be a problem if we want to harvest some (domain) knowledge from Java source code?

This work has been supported by the NWO TOPGO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”
```java
public static void testExample() {
    String x = "foo";
    Object staticResult = x.concat(x);
    Object dynResult = example("java.lang.String","concat", "foo",
            Collections.singletonMap("example", x), "example",
            Collections.singletonMap("example", x), "example"
            );

    assert staticResult.equals(dynResult);
}

public static Object example(String cln, String mn, Object init, Map<String, Object> map) throws InstantiationException, IllegalAccessException, ClassNotFoundException, NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException {
    try {
        Class<?> cl = Class.forName(cln);

        Object i = cl.getConstructor(init.getClass()).newInstance(init);

        Method m = cl.getMethod(mn, map.get("key").getClass());

        return m.invoke(i, map.get("key"));
    } catch (InstantiationException | IllegalAccessException | ClassNotFoundException | NoSuchMethodException | SecurityException | IllegalArgumentException | InvocationTargetException e) {
        return null;
    }
}
```

[The Muppet Show]
I **know** no general solution exists in theory!

You are both **biased** we could use empirical evidence…

But!! I can design tools which **work** on these categories

---


---

**Research Question:** What are limits of state-of-the-art static analysis tools when confronted with the Reflection API and how do these limits relate to real Java code?
Actionable results

• Researchers: high impact suggestions
• Practitioners: adapt code for robustness

Empirical evidence

• Complex reflection is everywhere in Java
• 462 Java projects in a representative and clean corpus
• 78% of Java projects have hard reflective code
• Known limitations have significant impact (4% - 54%)
• Existing soundy assumptions validated, more assumptions motivated

Answers to research questions

1. What is Java reflection?
2. How often is Java reflection used, and how?
3. What do static analysis tools do to resolve reflection?
4. What are limitations of static analysis tools?
5. How often does real Java code challenge limitations of static analysis?
Q1: What is Java reflection?

```java
public static void testExample() {
    String x = "foo";
    Object staticResult = x.concat(x);
    Object dynResult = example("java.lang.String","concat", "foo",
                               Collections.singletonMap("example", x), "

    assert staticResult.equals(dynResult);
}

public static Object example(String cln, String mn, Object init, Map<String, String> map) {
    try {
        Class<?> cl = Class.forName(cln);
        Object i = cl.getConstructor(init.getClass()).newInstance(init);
        Method m = cl.getMethod(mn, map.get(key).getClass());
        return m.invoke(i, map.get(key));
    }
    catch (InstantiationException | IllegalAccessException | NoSuchMethodException | SecurityException | InvocationTargetException e) {
        return null;
    }
}
```

"Hard" or "Easy"?
```
<Class> ::= 
  Class.forName(String)
  | Class.forName(String, Boolean, ClassLoader)
  | <ClassLoader>.loadClass(String)
  | <Type>.class
  | <Object>.getClass()
  | <Class>.getInterfaces()
  | <Class>.getSubClasses()
  | <Class>.getEnclosingMethod()

Constructor ::= 
  <Class>.getDeclaredConstructors()
  | <Class>.getDeclaredConstructor(Class)
  | <Class>.getEnclosingConstructor()

<String> ::= 
  <MetaObject>.getName()
  | <MetaObject>.toStringLength
  | <Class>.getPackage() // not shown
```
Q2: How often is reflection used?

- Corpus of 461 (out of 3000) OSS Java projects:
  - Maximize representativeness [55]
  - Clean [clone detection]
  - Parse & resolve [Rascal, Eclipse JDT]
  - Categorize [see Q1]

SWAT - SoftWare Analysis And Transformation

- Lookup Class (LC)
- Lookup Meta Object (LM)
- Traverse Meta Object (TM)
- Construct Object (C)
- Proxy (P)
- Access Object (A)
- Manipulate Object (M)
- Manipulate Meta Object (MM)
- Invoke Method (I)
- Array (AR)
- Cast (DC)
- Signature (SG)
- Assertions (AS)
- Annotations (AN)
- String representation (ST)
- Resource (RS)
- Security (S)

of projects using reflection
Q3: What do analysis tools do?

- Extended structured literature review
- 4K pdf’s
- Semi-automatic full text analysis
- Filtering from 4k via 514, to 50 to 33 pdf’s
- Annotating
- Categorizing
### Table III

Static Analysis approaches for handling reflection. For object and context sensitivity we report the sensitivity depth. For the strings column: ○ no analysis, ◦ only literals, ◈ literals and concatenations, and ◊ full fledged (JSA) string operations. For the remaining properties we use filled circles to summarize the coverage of a property: ○ for none, ◦ for partial, and ◊ for full. The table is sorted on the ‘Build using’ and ‘Year’ columns.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Year</th>
<th>Tool</th>
<th>Related</th>
<th>Kind</th>
<th>Goal</th>
<th>Sensitivity&lt;sup&gt;(y)&lt;/sup&gt;</th>
<th>Interprocedural</th>
<th>Fixed-point</th>
<th>Strings</th>
<th>Casts</th>
<th>Meta-Objects</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>2005</td>
<td>bddbddd</td>
<td>Static &amp; Annotations</td>
<td>Call Graph&lt;sup&gt;(a)&lt;/sup&gt;</td>
<td>○ ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>(k)</td>
<td>●</td>
<td>Datalog &amp; bddbddd</td>
</tr>
<tr>
<td>[4]</td>
<td>2009</td>
<td>Doop</td>
<td>[1, 5]</td>
<td>Static Points to Points to</td>
<td>○ ○ 0 1,2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>(c)</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>[6]</td>
<td>2013</td>
<td>Datalude</td>
<td>[1]</td>
<td>Static Points to Points to</td>
<td>○ ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>Maude &amp; Joeq</td>
</tr>
<tr>
<td>[7]</td>
<td>2014</td>
<td>ELF</td>
<td>[4]</td>
<td>Static Points to Points to</td>
<td>○ ○ 0 1,2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>Doop</td>
</tr>
<tr>
<td>[8]</td>
<td>2015</td>
<td>SOLAR</td>
<td>[7]</td>
<td>Static &amp; Annotations Points to Points to</td>
<td>○ ○ 0 1,2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>(d)</td>
<td>Doop &amp; ELF</td>
</tr>
<tr>
<td>[9]</td>
<td>2015</td>
<td></td>
<td>Static</td>
<td>Points to Points to</td>
<td>○ ○ 0 1</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>Datalog</td>
</tr>
<tr>
<td>[10]</td>
<td>2015</td>
<td>Doop</td>
<td>[4]</td>
<td>Static Points to Points to</td>
<td>○ ○ 0 1,2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>(e)</td>
<td>(e)</td>
<td>Datalog</td>
</tr>
<tr>
<td>[11]</td>
<td>2003</td>
<td>JSA</td>
<td>Static</td>
<td>Call Graph</td>
<td>○ ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>Soot</td>
</tr>
<tr>
<td>[12]</td>
<td>2007</td>
<td></td>
<td>Static &amp; Dynamic Class Loading</td>
<td>(b) (f) 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>(g)</td>
<td>○</td>
<td>Soot &amp; JSA</td>
<td></td>
</tr>
<tr>
<td>[13]</td>
<td>2009</td>
<td></td>
<td>Static &amp; Dynamic Class Loading</td>
<td>(b) (f) 0 0</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>(e)</td>
<td>○</td>
<td>Soot &amp; JSA</td>
<td></td>
</tr>
<tr>
<td>[14]</td>
<td>2013</td>
<td>AVERROG</td>
<td>Static &amp; Dynamic Modeling API</td>
<td>○ ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>Soot &amp; TamiFlex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[15]</td>
<td>2007</td>
<td>ACE</td>
<td>Static &amp; Dynamic Call Graph</td>
<td>○ ○ 0 1</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[16]</td>
<td>2011</td>
<td>Stowaway</td>
<td>Static</td>
<td>Name</td>
<td>○ ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>[17]</td>
<td>2012</td>
<td>Scandals</td>
<td>Static</td>
<td>Taint</td>
<td>○ ○ 0 1</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td></td>
</tr>
<tr>
<td>[18]</td>
<td>2013</td>
<td></td>
<td>Static</td>
<td>Name</td>
<td>(b) ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(i)</td>
<td></td>
</tr>
<tr>
<td>[19]</td>
<td>2014</td>
<td></td>
<td>Static</td>
<td>CFG</td>
<td>○ ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(i)</td>
<td></td>
</tr>
<tr>
<td>[20]</td>
<td>2014</td>
<td>PUSE</td>
<td>Static</td>
<td>Points to Points to</td>
<td>(b) ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(k)</td>
<td></td>
</tr>
<tr>
<td>[21]</td>
<td>2015</td>
<td>WALA</td>
<td>Static</td>
<td>Multiple</td>
<td>(b) ○ 0 0</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(k)</td>
<td>Checker Framework</td>
</tr>
<tr>
<td>[22]</td>
<td>2015</td>
<td>part of SPARTA</td>
<td>Static &amp; Annotations Implicit CFG</td>
<td>○ ○ 0 0</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(j)</td>
<td>dx</td>
<td></td>
</tr>
<tr>
<td>[24]</td>
<td>2015</td>
<td>EdgeMiner</td>
<td>Static</td>
<td>Implicit CFG</td>
<td>○ ○ 0 0</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>(j)</td>
<td></td>
</tr>
</tbody>
</table>

---

a) Including points-to analysis.
b) After SSA transform.
c) Only for Class.forName.
d) Lazy.
e) Only if it points to a small set of candidates (subclasses / fields / methods).
f) Only string fields.
g) JSA extended with environment information, modeling field, and tracking of objects of type Object.
h) Backwards slicing.
i) With heuristics.
j) Only for bus (JRE/Android) framework.
k) Only for newInstance.
l) None of the papers are path sensitive.
m) The reported flow sensitivity was always intra-procedural.
Table III

For object and context sensitivity we report the sensitivity depth. For the strings concatenations, and full fledged (JSA) string operations. For the remaining property: ○ for none, ● for partial, and ● for full. The table is sorted on the “Build up” and “Year” columns.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Sensitivity&lt;sup&gt;(y)&lt;/sup&gt;</th>
<th>Interprocedural</th>
<th>Fixed-point</th>
<th>Strings</th>
<th>Casts</th>
<th>Meta-Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>flow&lt;sup&gt;(z)&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>object</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>context</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Graph&lt;sup&gt;(a)&lt;/sup&gt;</td>
<td>○  ○  0  0</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0  1, 2</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0  1, 2</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  1  1</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0  1, 2</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Points to</td>
<td>○  ○  ○  0  1, 2</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

Day 1

Day 2

Day 3

Day 4

Day 5

Day 6

Day 7

Day 8

Day 9

Day 10
Q4: What are the limitations? and Q5: how do these relate to real code?

- Collect and categorize analysis papers self-reported:
  - Optimistic ‘soundy’ assumptions about code
  - Known limitations of the algorithms
  - What is their damage in the corpus?
- Method:
  - Recognize and count counter examples
  - Applying AST patterns to the entire corpus
  - Rascal metaprogramming language

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Impact</th>
<th>Precision</th>
<th>Code intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CorrectCasts</td>
<td>4%</td>
<td>8/10</td>
<td>Supplying a fallback or looping through candidates and swallowing the exception</td>
</tr>
<tr>
<td>Ignoring-Exceptions1</td>
<td>23%</td>
<td>10/10</td>
<td>Falling back to a less specific Meta Object, or switching to a different ClassLoader</td>
</tr>
<tr>
<td>Ignoring-Exceptions2</td>
<td>38%</td>
<td>9/10</td>
<td>Iterating through candidates and either breaking when one does not throw an exception, or continuing to the next candidates</td>
</tr>
<tr>
<td>Inaccurate-Indexed-Collectors</td>
<td>55%</td>
<td>exact</td>
<td>Iterating through a signature of an meta object</td>
</tr>
<tr>
<td>Inaccurate-AndMaps</td>
<td>55%</td>
<td>exact</td>
<td>Meta objects as function pointers in a table, mapping to objects, caching around Reflection API</td>
</tr>
<tr>
<td>NoMultiple-MetaObjects</td>
<td>54%</td>
<td>exact</td>
<td>Looking through candidates, performing mass updates of fields, checking signatures</td>
</tr>
<tr>
<td>Ignoring-Environment</td>
<td>2%</td>
<td>10/10</td>
<td>Only 9 instances found, they were all dependency injection</td>
</tr>
<tr>
<td>Undecidable-Filtering</td>
<td>48%</td>
<td>8/10</td>
<td>Trying different names of meta objects, filtering method and fields based on signature</td>
</tr>
<tr>
<td>NoProxy</td>
<td>21%</td>
<td>exact</td>
<td>Wrapping objects for caching or transactions, automatically converting between comparable interfaces</td>
</tr>
</tbody>
</table>
Suggestions for static analysis researchers and Java language designers

1. Reflection API improvements to restrict arbitrary interactions (i.e. using lambdas)
2. Infer information from downcasts more aggressively
3. Make soundy assumptions about dynamic proxies: the “oblivious wrapper proxy”
4. Model common “goto patterns” with exceptions around reflection
5. Soundly assume boundedness and unorderedness of meta object collections
6. Apply dynamic language analysis techniques to methods which have reflection

Advice for software engineers; make your code more robust now

1. Do not factor reflection into type polymorphic methods
2. Never use dynamic proxies
3. Use local variables/fields for meta object storage
4. Avoid loops over collections of meta objects
5. Test for preconditions instead of waiting for exceptions
Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study

@davylanman @aserebrenik @jurgenvinju

Please use these artefacts for yourselves, or contact us for discussion about:
- the new soundy assumptions are a prioritized work list (*)
- the corpus is a way to validate relevance for new ideas in static analysis [3]
- tell us why we were wrong (replicate it) [63]


To the authors of the static analysis papers, to the anonymous reviewers and to the members of IFIP WG 2.4 Software Implementation Technology, including Anders Møller

This work has been supported by the NWO TOPCO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”