A Fine-Grained Debugger for Aspect-Oriented Programming

Haihan Yin, Christoph Bockisch, Mehmet Akşit

{h.yin, c.m.bockisch, m.aksit}@cs.utwente.nl

TRESE Software Engineering group
University of Twente, The Netherlands
http://trese.cs.utwente.nl
Why Debug AO Programs?

- Aspects can implicitly alter structure and behavior of other modules.
- Case study: 42 out of 102 AOP-related bugs are because of implicitness.
  - F. Ferrari et.al. An exploratory study of fault-proneness in evolving aspect-oriented programs. ICSE'10
- Debuggers are used for analyzing and locating faults.

### Diagram

- **Bug**
  - has **Fault**
    - causes **Symptom**
      - locates **Analysis**
        - guides **Observation**
          - observes **Symptom**
            - has **Bug**

- **Runtime tools** e.g. Debuggers
  - performs by **Programmer**

Case study: 42 out of 102 AOP-related bugs are because of implicitness.

- F. Ferrari et.al. An exploratory study of fault-proneness in evolving aspect-oriented programs. ICSE'10
Limitations of Existing Debuggers

- AO programs are compiled to intermediate-representation (IR) of the base language
- IR partially or even entirely loses AO-information. In AspectJ,

- Debuggers that use debugging information stored in IR
  - Java Debugger (JDB)
  - AODA – W. De Borger et al. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. AOSD'09
  - Wicca – M. Eaddy et al. Debugger aspect-enabled programs. SC'07
  - TOD – G. Pothier and E. Tanter. Extending omniscient debugging to support aspect-oriented programming. SAC'08
Fault Categories & Symptoms

- **Pointcut related**
  - Incorrect pointcut composition
  - Incorrect pattern
  - Incorrect designator
  - Incorrect context

  An advice is unexpectedly executed or not executed

- **Advice related**
  - Incorrect composition control
  - Incorrect flow change
  - Violated requirements

  An advice does not behave as expected
Observations & Debugging Tasks

Analysis \(\leftarrow\) guides \(\rightarrow\) Observation

- Setting AO breakpoint
- Evaluating pointcut sub-expressions
- Evaluating pattern sub-expressions
- Flattening pointcut references
- Inspecting runtime values
- Inspecting AO-conforming stack traces
- Inspecting program compositions
- Inspecting precedence dependencies
- Excluding and adding AO definitions

An advice is unexpectedly executed or not executed

An advice does not behave as expected

Haihan Yin \{h.yin@cs.utwente.nl\}

23/03/2012
Aspect-Oriented Intermediate Representation

Source code:

```java
aspect Azpect {
    before Base b) :
        call(* Base.foo()) &&
        target(b)
    { ... }
}
```

AO IR in form of XML:

```xml
<attachment>
    <specialization>
        ...
        <atomicPredicate type="InstanceofPredicate">
            <requiredTypeName location=... >Base</requiredTypeName>
            <context type="CalleeContext" location=... />
        </atomicPredicate>
        <context type="CalleeContext" location=... />
        ...
    </specialization>
</attachment>
```
The Debugger Architecture

Aspects

Byte Code

AO IR

compile

reflect

compile

import

load

ADDI Model

User Interface

Debugger VM

Execution Environment

Debuggee VM

JPDA

Haihan Yin {h.yin@cs.utwente.nl}
Advanced-Dispatching Debug Interface

Evaluating pattern sub-expression
Inspecting precedence dependencies
Excluding and adding AO definitions
Inspecting program compositions

UNIVERSITY OF TWENTE.
The Graphical Representation of Dispatch

1. public class Base {
2.   public static void main(String[] args) {
3.     Base b = new BaseSub();
4.     b.advicedMethod();
5.   }
6.   public void advicedMethod() {
7.     // ...
8.   }
9. }
10. public class BaseSub extends Base { // ...

11. public aspect Azpect {
12.   pointcut base() : call(* Base.advicedMethod());
13.   before() : base() && !target(BaseSub) {
14.     // ...
15.   }
16.   Object around() : base() {
17.     proceed();
18.     return null;
19.   }
20. }

Bug: Wrong negation operator
The Graphical Representation of Precedence Dependencies

```java
1 aspect PrecedingAspect {
2   before(): call(* Base.advisedMethod()) { //... }
3 }
4 aspect PrecededAspect {
5   before(): call(* Base.advisedMethod()) { //... }
6 }
7 aspect PrecedenceAspect {
8   declare precedence: PrecededAspect, PrecedingAspect;
9 }
10 public aspect Aspect {
11   pointcut base(): call(* Base.advisedMethod());
12   before(): base() && target(BaseSub) { //... }
13   Object around(): base() {
14     proceed();
15     return null; }
16 }
```

Bug: Wrong precedence
Summary

7 AO-fault categories
Pointcut related
- Incorrect pointcut composition
- Incorrect pattern
- Incorrect designator
- Incorrect context
Advice related
- Incorrect composition control
- Incorrect flow change
- Violated requirements

10 Tasks
Setting AO breakpoint
- Evaluating pointcut sub-expressions
- Evaluating pattern sub-expressions
- Flattening pointcut references
- Inspecting runtime values
- Inspecting AO-conforming stack traces
- Inspecting program compositions
- Inspecting precedence dependencies
- Excluding and adding AO definitions
- Locating AO constructs

 Fault causes Symptom
 Fault locates Observation
 Analysis guides Observation

Haihan Yin {h.yin@cs.utwente.nl}