Change Impact Analysis for AspectJ Programs

Sai Zhang, Zhongxian Gu, Yu Lin and Jianjun Zhao

Shanghai Jiao Tong University
Change Impact Analysis for AspectJ Programs

• AspectJ’s specific constructs requires adapting the existing analysis techniques
  – Requires to handle the unique aspectual features

• Can we develop techniques/tools automatically determine the affected program fragments, affected tests and their responsible changes?

• In this paper, we present an approach to address both questions with atomic change and AspectJ call graph representation
Outline

• **Background and Motivation**
  – Software Change impact analysis
  – AspectJ semantics and analysis challenges

• **Contributions**
  – A catalog of atomic changes for AspectJ, to capture semantic change information
  – A change impact analysis model for AspectJ programs
  – Experimental Evaluation

• **Conclusion**
  – Change impact analysis applications
  – Future work
Outline

• **Background and Motivation**
  – Software Change impact analysis
  – AspectJ semantics and analysis challenges

• **Contributions**
  – A catalog of atomic changes for AspectJ, to capture semantic change information
  – A change impact analysis model for AspectJ programs
  – Experimental Evaluation

• **Conclusion**
  – Change impact analysis applications
  – Future work
Software Change Impact Analysis

• A useful technique for software evolution, it can be used to:
  • Determine the effects of a source editing session, including:
    • Predict the potential impact of changes before applied
    • Estimate the side-effect of changes after they are addressed

• Applications of change impact analysis
  • Testing, debugging, change assessment, etc.
AspectJ Semantic

• An AspectJ program can be divided into two parts:
  – *Base code*, that is, language constructs as in Java
  – *Aspect code*, includes aspectual constructs, like *join point*, *pointcut*, *advice*, *intertype declarations*.

• A Simple Example:

  ```java
  aspect M {
    pointcut callPoints(): execution(* C.n());
    after(): callPoints() { .... }
  }
  
  class C {
    void n(){...}
  }
  ```

  Join point
An AspectJ program can be divided into two parts:

- **Base code**, that is, language constructs as in Java
- **Aspect code**, includes aspectual constructs, like *join point*, *pointcut*, *advice*, *intertype declarations*.

A Simple Example:

```java
aspect M {
    pointcut callPoints():
        execution(* C.n());
    after(): callPoints() { .... }
}

class C {
    void n(){...}
}

main() {
    new C().n()
    ...
}
```

invoke
Analyses Challenges

• Changes in both aspect/base code can change dramatically the program behavior
  – Such as editing pointcut designator

• Can we directly apply existing techniques to AspectJ programs?
  – The discrepancy between source code and the woven bytecode can be significant
  – Compiler-specific code
  – Hard to estimate relationships for mapping analysis result to the source code [Xu et al. ICSE 07]

• A more general question
  – What is an appropriate static change representation for AspectJ software for impact analysis and other tasks?
Our approach

• Perform *source-code-level static analysis* for AspectJ software

• Use *atomic changes* to represent code modifications in AspectJ program (extend Ryder et al. OOPSLA 04’s catalog for Java)

• Employ *static aspect-aware call graph* to safely identify impacted program fragments
Outline

• Background and Motivation
  – Software Change impact analysis
  – AspectJ semantics and analysis challenges

• Contributions
  – A catalog of atomic changes for AspectJ, to capture semantic change information
  – A change impact analysis model for AspectJ programs
  – Experimental Evaluation

• Conclusion
  – Change impact analysis applications
  – Future work
Atomic Change Representation

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Atomic Change Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Add an Empty Aspect</td>
</tr>
<tr>
<td>DA</td>
<td>Delete an Empty Aspect</td>
</tr>
<tr>
<td>INF</td>
<td>Introduce a New Field</td>
</tr>
<tr>
<td>DIF</td>
<td>Delete an Introduced Field</td>
</tr>
<tr>
<td>CIF</td>
<td>Change an Introduced Field Initializer</td>
</tr>
<tr>
<td>INM</td>
<td>Introduce a New Method</td>
</tr>
<tr>
<td>DIFM</td>
<td>Delete an Introduced Method</td>
</tr>
<tr>
<td>CIMB</td>
<td>Change an Introduced Method Body</td>
</tr>
<tr>
<td>AED</td>
<td>Add an Empty Advice</td>
</tr>
</tbody>
</table>

Reflects the semantic different between the original program \(P\) and edited program \(P'\), in forms of \(<\text{joinpoint}, \text{advice}>\) matching tuples.

The formal definition of \(AIC\) is shown as follows:

\[
AIC = \{ <j, a> \mid <j, a> \in ((J' \times A' - J \times A) \cup (J \times A - J') \times A') \}
\]

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Advice Invocation Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAA</td>
<td>Delete an Aspect Joinpoint</td>
</tr>
<tr>
<td>ASED</td>
<td>Add a Soften Exception Declaration</td>
</tr>
<tr>
<td>DSED</td>
<td>Delete a Soften Exception Declaration</td>
</tr>
<tr>
<td>AIC</td>
<td>Advice Invocation Change</td>
</tr>
</tbody>
</table>

Table 1: A catalog of atomic changes in AspectJ
aspect M {
  pointcut callPoints():
    execution(* C.n());
  after(): callPoints() { .... }
}

class C {
  void n(){...
  }
}

AA(M)
ANP(callPoints), CPB(callPoints)
AEA(after: callPoints), CAB(after: callPoints)
AIC(C.n(), after: callPoints)
Inter-dependences between atomic changes

• **Syntax dependence**
  – To ensure the syntactical correctness of program when applying one change

• **Interaction dependence**
  – Model the interactions between aspect code and base code

• **Why we need dependence?**
  – Capture semantic relationships between source code change
  – Construct intermediate program versions for debugging
  – Use for further analysis, such as incremental analysis
aspect M {
    
    pointcut callPoints():
        execution(* C.n());

    after(): callPoints() { .... }
}

class C {
    
    void n(){...}
}

AA (M)
ANP( callPoints), CPB( callPoints)

AEA( after: callPoints), CAB( after: callPoints)
AIC( C.n(), after: callPoints)

CAB depends on AEA => CAB ⇐ AEA
AEA depends on ANP => AEA ⇐ ANP
Example: Interaction Dependence

aspect M {
  
  pointcut callPoints():
    
  execution(* C.n());

  after(): callPoints() {
    .... }

}


class C {
  
  void n(){...}

}

AA (M)
ANP( callPoints) , CPB(callPoints)
AEA( after:callPoints) , CAB(after:callPoints)
AIC(C.n(), after:callPoints)

AIC depends on AEA  =>  AIC  < AEA
Change Impact Analysis Model

• A change impact analysis model for AspectJ programs
  – Used to identify affected program fragments, affected regression tests, and their corresponding changes

• This analysis model is based on *aspect-aware* call graph
  – Use RTA algorithm to build static call graph for the base code
  – Treat advice as a method-like node
  – Matching relationship of <advice, joinpoint> as edges
  – Finally connect base code and aspect code graph
    • Conservative assumption for dynamic pointcut
aspect M {
    pointcut callPoints():
        execution(* C.n());
    after(): callPoints() {
        ...
    }
}

class C {
    void n() {
        ...
    }
}

class Tests {
    void testN() {
        new C().n();
    }
}

Example: call graph

Tests.testN() → C.n() → after():callPoints
Impact Analysis Model

• Detecting affected program fragments
  – Traversing the call graph from the modified nodes

• Detecting affected tests
  – The call graph of test contains an affected node

• Detecting responsible changes
  – All the atomic changes appearing on the call graph nodes (edges), and all their prerequisites
We implement our automatic analysis tool, Celadon, on top of *ajc compiler* [ICSE 08 demo, AOSD 08 demo]
Evaluation

- Evaluation and applications
  - On 24 AspectJ benchmark versions

- Subject programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>#Loc</th>
<th>#Ver</th>
<th>#Me</th>
<th>#Shad</th>
<th>#Tests</th>
<th>%mc</th>
<th>%asc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>111</td>
<td>3</td>
<td>18</td>
<td>15</td>
<td>27</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Figure</td>
<td>147</td>
<td>4</td>
<td>23</td>
<td>5</td>
<td>20</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Bean</td>
<td>199</td>
<td>3</td>
<td>12</td>
<td>8</td>
<td>15</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Tracing</td>
<td>1059</td>
<td>4</td>
<td>44</td>
<td>32</td>
<td>15</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>NullCheck</td>
<td>2991</td>
<td>4</td>
<td>196</td>
<td>146</td>
<td>128</td>
<td>96.9</td>
<td>85.8</td>
</tr>
<tr>
<td>Lod</td>
<td>3075</td>
<td>2</td>
<td>220</td>
<td>1103</td>
<td>157</td>
<td>90.0</td>
<td>63.4</td>
</tr>
<tr>
<td>Dcm</td>
<td>3423</td>
<td>2</td>
<td>249</td>
<td>359</td>
<td>157</td>
<td>94.3</td>
<td>73.5</td>
</tr>
<tr>
<td>Spacewar</td>
<td>3053</td>
<td>2</td>
<td>288</td>
<td>369</td>
<td>132</td>
<td>88.5</td>
<td>74.0</td>
</tr>
</tbody>
</table>
Experimental Result (1)

- Atomic changes between version pairs

*Celadon successfully handle aspectual features.*
## Experimental Result (2)

- Affected tests and affecting changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Total Number</th>
<th>% at</th>
<th>% ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>24</td>
<td>100%</td>
<td>67%</td>
</tr>
<tr>
<td>Q3</td>
<td>38</td>
<td>100%</td>
<td>71%</td>
</tr>
<tr>
<td>F2</td>
<td>22</td>
<td>60%</td>
<td>55%</td>
</tr>
<tr>
<td>F3</td>
<td>80</td>
<td>80%</td>
<td>58%</td>
</tr>
<tr>
<td>F4</td>
<td>59</td>
<td>30%</td>
<td>17%</td>
</tr>
<tr>
<td>B2</td>
<td>35</td>
<td>80.0%</td>
<td>86%</td>
</tr>
<tr>
<td>B3</td>
<td>11</td>
<td>40%</td>
<td>100%</td>
</tr>
<tr>
<td>T2</td>
<td>41</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>69</td>
<td>100%</td>
<td>48%</td>
</tr>
<tr>
<td>T4</td>
<td>37</td>
<td>100%</td>
<td>73%</td>
</tr>
<tr>
<td>N2</td>
<td>35</td>
<td>78%</td>
<td>89%</td>
</tr>
<tr>
<td>N3</td>
<td>7</td>
<td>78%</td>
<td>86%</td>
</tr>
<tr>
<td>N4</td>
<td>2</td>
<td>51%</td>
<td>100%</td>
</tr>
<tr>
<td>L2</td>
<td>1979</td>
<td>100%</td>
<td>75%</td>
</tr>
<tr>
<td>D2</td>
<td>85</td>
<td>86%</td>
<td>67%</td>
</tr>
<tr>
<td>S2</td>
<td>74</td>
<td>30%</td>
<td>85%</td>
</tr>
</tbody>
</table>
### Experimental Result (3)

- **Affected program fragment (at method level)**

<table>
<thead>
<tr>
<th>Version</th>
<th>Nodes Num</th>
<th>Affected Nodes</th>
<th>% Affected Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>22</td>
<td>12</td>
<td>55%</td>
</tr>
<tr>
<td>Q3</td>
<td>23</td>
<td>13</td>
<td>57%</td>
</tr>
<tr>
<td>F2</td>
<td>26</td>
<td>5</td>
<td>19%</td>
</tr>
<tr>
<td>F3</td>
<td>32</td>
<td>17</td>
<td>53%</td>
</tr>
<tr>
<td>F4</td>
<td>74</td>
<td>24</td>
<td>32%</td>
</tr>
<tr>
<td>B2</td>
<td>73</td>
<td>24</td>
<td>33%</td>
</tr>
<tr>
<td>B3</td>
<td>45</td>
<td>14</td>
<td>31%</td>
</tr>
<tr>
<td>T2</td>
<td>112</td>
<td>22</td>
<td>20%</td>
</tr>
<tr>
<td>T3</td>
<td>112</td>
<td>22</td>
<td>20%</td>
</tr>
<tr>
<td>T4</td>
<td>118</td>
<td>12</td>
<td>11%</td>
</tr>
<tr>
<td>N2</td>
<td>708</td>
<td>677</td>
<td>96%</td>
</tr>
<tr>
<td>N3</td>
<td>709</td>
<td>683</td>
<td>96%</td>
</tr>
<tr>
<td>N4</td>
<td>709</td>
<td>126</td>
<td>18%</td>
</tr>
<tr>
<td>L2</td>
<td>759</td>
<td>705</td>
<td>93%</td>
</tr>
<tr>
<td>D2</td>
<td>851</td>
<td>382</td>
<td>45%</td>
</tr>
<tr>
<td>S2</td>
<td>1162</td>
<td>446</td>
<td>38%</td>
</tr>
</tbody>
</table>

For change assessment
Experiment Discussion

• Discussion
  – Promising experimental result for AspectJ programs
  – Handle aspectual features

• Threats to validity
  – Scalability
  – Human bias
Outline

• Background and Motivation
  – Software Change impact analysis
  – AspectJ semantics and analysis challenges

• Contributions
  – A catalog of atomic changes for AspectJ, to capture semantic change information
  – A change impact analysis model for AspectJ programs
  – Experimental Evaluation

• Related Work

• Conclusion
  – Change impact analysis applications
  – Future work
Related Work

- Atomic Changes in OO Programs [Ryder et al 01]
- Change Impact Analysis for Java [Ren et al 04]
- Change Impact Analysis for AspectJ [Zhao 02, Shinomi et al 05, Stoerzer 05]
- Change Impact Analysis Applications [Chelsey 05, Ren 06, 07, Stoerzer 06]
- Regression Tests Selection [Zhao 06, Xu 07]
- Delta Debugging [Zeller et al, 99, 02, 05, Mishergi et al 06]
Outline

• Background and Motivation
  – Software Change impact analysis
  – AspectJ semantics and analysis challenges

• Contributions
  – A catalog of atomic changes for AspectJ, to capture semantic change information
  – A change impact analysis model for AspectJ programs
  – Experimental Evaluation

• Conclusion
  – Change impact analysis applications
  – Future work
Limitation and Future Work

• Improve the visualization of output result
  – Rich information instead of a textual tree-based representation
  – More clearly for programmer’s to use

• Improve the atomic change model for AO programs
  – Modeling dynamic pointcut, like cf low

• Investigate more applications
  – Automated debugging support [PASTE 08]
  – Maintainability assessment [TASE 08]
  – Incremental analysis [Technical report]
  – ...

Summary

• We extend the atomic changes in Java to AspectJ programming language.

• We present a change impact analysis model for AspectJ programs.

• We implement Celadon, a change impact analysis tool for AspectJ Programs.

• We apply Celadon to other program analysis applications, such as automatic debugging.