Incremental Call Graph Reanalysis for AspectJ Software

Yu Lin, Sai Zhang and Jianjun Zhao

Shanghai Jiao Tong University, China
Outline

• Motivation and Background
  – Incremental analysis of call graph
  – AspectJ semantics and analysis challenges
  – Atomic changes and call graph for AspectJ

• Our incremental approach
  – Atomic change classification
  – Relationship between atomic change and call graph
  – Incremental algorithm of constructing call graph for AspectJ
  – Experimental evaluation

• Conclusion and future work
Motivation

• Call graph construction is a key task required by many approaches to whole program optimization and understanding
  – Compiler optimization and program comprehension
  – Software maintenance tasks

• During software evolution, program change is an essential operation

• The call graph representation for the whole program may change
  – The combination of small changes may affect the whole call graph

• Most call graph analysis approaches are *global exhaustive analysis*
  – Analyze the whole program code to construct the call graph
  – It may be unnecessary to recompute a call graph when change occurs
For aspect-oriented program, aspectual features complicate the analysis

- **base code** (e.g. classes, methods in Java), and **aspect code** (e.g. aspects, advices which are modeling crosscutting concerns in the program)

```java
aspect M { aspect

    pointcut callPoints():

    execution(* C.n());

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

}
```
• For aspect-oriented program, aspectual features complicate the analysis
  – *base code* (e.g. classes, methods in Java), and *aspect code* (e.g. aspects, advices which are modeling crosscutting concerns in the program)

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

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

- For aspect-oriented program, aspectual features complicate the analysis
  - *base code* (e.g. classes, methods in Java), and *aspect code* (e.g. aspects, advices which are modeling crosscutting concerns in the program)

```
aspect M {
  class C {
    void n(){...}
  }

  pointcut callPoints():
    execution(* C.n());

  advice
    after(): callPoints() { .... }
}
```
AspectJ

- For aspect-oriented program, aspectual features complicate the analysis
  - *base code* (e.g. classes, methods in Java), and *aspect code* (e.g. aspects, advices which are modeling crosscutting concerns in the program)

```java
class C {
    void n() {
        // ...}
}

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

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

main() {
    new C().n();
    // ... }
```
• For aspect-oriented program, aspectual features complicate the analysis
  – *base code* (e.g. classes, methods in Java), and *aspect code* (e.g. aspects, advices which are modeling crosscutting concerns in the program)
    
  – Aspect code changes dramatically the behavior of the base code as well as its call graph structure
    
  – E.g. without any change to the base code, an aspect can change the calling relationship between the call graph nodes
Our paper focuses on how to incrementally compute the new call graph for AspectJ programs.

**Basic Idea:**
- First, decompose the source changes between two program versions into a set of *atomic changes* (utilize our previous framework *Celadon*)
- Then, exploit the relationship between atomic changes and AspectJ call graph, to update the initially constructed graph.
Incremental Call Graph Reanalysis for AspectJ Software

• A source-code-level (static) call graph construction approach

• Reuses results from previous analysis (i.e. atomic changes) to update the call graph

• Performs an amount of work proportional to the source changes
  – A large portion of unnecessary reanalysis can be avoided
Atomic Changes for AspectJ

- **A catalog of atomic changes for AspectJ**
  - Abstract change representation
  - Proposed for capturing semantic change information, which is the foundation of our reanalysis
  - Dynamic programming algorithm (Wuu Yang 1991) to compare ASTs, for generating atomic changes
## Atomic Changes for AspectJ

<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>CIFI</td>
<td>Change an Introduced Field Initializer</td>
</tr>
<tr>
<td>INM</td>
<td>Introduce a New Method</td>
</tr>
<tr>
<td>DIM</td>
<td>Delete an Introduced Method</td>
</tr>
<tr>
<td>CIMB</td>
<td>Change an Introduced Method Body</td>
</tr>
<tr>
<td>AEA</td>
<td>Add an Empty Advice</td>
</tr>
<tr>
<td>DEA</td>
<td>Delete an Empty Advice</td>
</tr>
<tr>
<td>CAB</td>
<td>Change an Advice Body</td>
</tr>
<tr>
<td>ANP</td>
<td>Add a New Pointcut</td>
</tr>
<tr>
<td>CPB</td>
<td>Change a Pointcut Body</td>
</tr>
<tr>
<td>DPC</td>
<td>Delete a Pointcut</td>
</tr>
<tr>
<td>AHD</td>
<td>Add a Hierarchy Declaration</td>
</tr>
<tr>
<td>DHED</td>
<td>Delete a Hierarchy Declaration</td>
</tr>
<tr>
<td>DAR</td>
<td>Delete an Aspect Proceed</td>
</tr>
<tr>
<td>ASED</td>
<td>Add a Softhen Exception Declaration</td>
</tr>
<tr>
<td>DSED</td>
<td>Delete a Softhen Exception Declaration</td>
</tr>
<tr>
<td>AIC</td>
<td>Advice Invocation Change</td>
</tr>
</tbody>
</table>

### Advice Invocation Change

- **AIC**

---

**self-explanatory**

- **Introduce a New Method**
- **Add an Empty Advice**
Atomic Changes for AspectJ

AIC | Advice Invocation Change

Reflects the semantic differences between the original program P and edited program P’, in the form of <joinpoint, advice> matching tuples

The formal definition of AIC is shown as follows:

$$AIC = \{ <j, a> \mid <j, a> \in ((J’ \times A’ - J \times A) \cup (J \times A - J’ \times A’)) \}$$
Example

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

```java
class C {
    void n() {
        ...
    }
}
```
Example

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

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

AA (M)
Example

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

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

AA (M)

ANP(callPoints), CPB(callPoints)
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)
Example

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)
Call Graph for AspectJ Software

- For base code
  - Employ CHA (Class Hierarchy Analysis) algorithm

- For aspect code
  - Consider the advice as a method-like node (advice node)
  - The advice invocations are represented by an edge (shadow edge) from the join point

- The complete call graph
  - Connect the call graph of aspect code to the call graph of base code, using the join point matching information
Call Graph for before and after advice

- **before** and **after** advice
  - Woven before or after the join points
  - Add shadow edges before or after the corresponding join point

```c
void m() { n() };
before(): call(*n(){ ... })
after(): execution(*n(){ ... })
```

Call site is a dummy node which means there is a call relationship between caller and callee, and is used to differentiate call join point and execution join point.
Call Graph for *around* advice

- **around** advice
  - Woven in place of the methods it matches
  - **proceed** call leads the execution back to the methods replaced by the around advice
  - Use two additional call sites to represent proceed call and proceed return edges

```java
void m() { n(); }
around(): call(* *.n()){ ...; proceed(); }
```

![Diagram showing call graph for around advice](image)
• Motivation and Background
  – Incremental analysis of call graph
  – AspectJ semantics and analysis challenges
  – Atomic changes and call graph for AspectJ

• Our incremental approach
  – Atomic change classification
  – Relationship between atomic change and call graph
  – Incremental algorithm of updating call graph for AspectJ
  – Experimental Evaluation

• Conclusion and future work
Atomic change classification

• The original atomic changes are not classified

• Why we need to classify atomic changes?
  – To facilitate the process of updating the call graph based on atomic changes

• How do we classify atomic changes?
  – According to the effect of each atomic change to the call graph
  – Five kinds of atomic changes
Method-related changes in *base* code:

<table>
<thead>
<tr>
<th>AM</th>
<th>Add an empty Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM</td>
<td>Delete an empty Method</td>
</tr>
<tr>
<td>CM</td>
<td>Change a Method Body</td>
</tr>
</tbody>
</table>

- **CM-Add**: add a new method call
- **CM-Del**: delete an existing method call
- **CM-Local**: do not affect the call graph

```c
void m() { ...; n(); n2(); }
```

- **CM-Add**:

```c
void m() { ...; n(); }
```

- **CM-Del**:  

```c
void m() { ...; n(); i++; }
```

- **CM-Local**:  

```c
void m() { ...; n(); i++; }
```
### Method-related changes

#### Method-related changes in base code:

<table>
<thead>
<tr>
<th>AM</th>
<th>Add an empty Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM</td>
<td>Delete an empty Method</td>
</tr>
<tr>
<td>CM</td>
<td>Change a Method Body</td>
</tr>
</tbody>
</table>

**CM-Add:** add a new method call  
**CM-Del:** delete an existing method call  
**CM-Local:** do not affect the call graph

#### Method-related changes in aspect code:

<table>
<thead>
<tr>
<th>INM</th>
<th>Introduce a New Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIM</td>
<td>Delete an Introduced Method</td>
</tr>
<tr>
<td>CIMB</td>
<td>Change an Introduced Method Body</td>
</tr>
</tbody>
</table>

**CIMB-Add**  
**CIMB-Del**  
**CIMB-Local**
Advice-related changes

- Advice-related changes are similar to method-related changes:

<table>
<thead>
<tr>
<th>AEA</th>
<th>Add an empty Advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEA</td>
<td>Delete an empty Advice</td>
</tr>
<tr>
<td>CAB</td>
<td>Change an Advice Body</td>
</tr>
</tbody>
</table>

- CAB-Add
- CAB-Del
- CAB-Local
Dynamic dispatch change

- **LC** (Lookup Change)
  - LC abstracts any kind of source edits that would affect dynamic dispatch behavior
  - E.g. adding or deleting methods, and adding or deleting inheritance relations
  - LC is represented by triple $\text{LookUp}: \langle \text{runtimeReceiverType}, \text{staticMethodSignature}, \text{actualMethodBound} \rangle$

```java
class A {
    void n() {
        // ...
    }
}
class B extends A {
    void n() {
        // ...
    }
}
class C extends B {
    // ...
}
```

```java
A a = new C();
a.n();
```

```
\langle C, A.n(), B.n() \rangle
```

- **Runtime Receiver Type**
- **Static Method Signature**
- **Actual Method Bound**
Dynamic dispatch change

- **LC (Lookup Change)**
  - LC abstracts any kind of source edits that would affect dynamic dispatch behavior
  - E.g. adding or deleting methods, and adding or deleting inheritance relations

- LC is represented by triple $\text{LookUp}: \langle \text{runtimeReceiverType}, \text{staticMethodSignature}, \text{actualMethodBound} \rangle$

- Preserve the precision of CHA algorithm by exploring the LC changes
Dynamic dispatch change

- We classify LC into two categories:
  - LC-Add and LC-Del
  - The LC changes will add or delete edges of the call graph

\[ \text{LookUp}_{\text{old}} : \]
LookUp triples in the old version

\[ \text{LookUp}_{\text{new}} : \]
LookUp triples in the new version
Dynamic dispatch change

• We classify LC into two categories:
  – LC-Add and LC-Del
  – The LC changes will add or delete edges of the call graph

\[ \text{LookUp}_{\text{preserve}} : \text{unchanged } \text{LookUp} \text{ triples} \]
Advice invocation change

- **AIC**
  - Contains information about advice invocations
  - Indicates that the advice invoking at the certain join point has been changed
  - Represented by tuple <joinpoint, advice>

![Venn Diagram showing AIC and unchanged tuples](image-url)
Advice invocation change

- We classify AIC into two categories:
  - AIC-Add and AIC-Del
  - The newly added invocations and deleted <joinpoint, advice> pairs
Ignored changes

- We ignore other changes (not mentioned above) that will not directly affect the call graph structure.

- Changes like CFI (Change definition of a instance Field Initializer) and AA (Add an Aspect) do not affect call graph.

- The effect of some changes like CPB (Change a Pointcut Body) can be reflected by AIC.
Use Atomic Change to Update Call Graph

• Match each change to the initial constructed call graph to decide which parts need to be updated

  • **AM, INM, and AEA change**
    – Add new nodes to the original call graph
  
  • **DM, DIM, and DEA change**
    – Delete existing nodes and their incoming/outgoing edges
  
  • **CM, CIMB, and CAB change**
    – Update corresponding method, intertype method, or advice nodes
    – Use the classification information (e.g. **CM-Add**) to find what kind of changes (e.g. adding call relations) have taken place
Example (1)

class A{
    void x(){
        y();
    }
    void y(){ }
}

class A{
    void x(){
        y();
        z();
    }
    void y(){ }
    void z(){ }
}

class A{
    void x(){
        y();
    }
    void y(){ }
}

AM(A.z)
CM-Del(A.x, A.y)
DM(A.y)
CM-Add(A.x, A.z)
Use Atomic Change to Update Call Graph

• **LC** changes
  – Update corresponding virtual call sites
  – For each triple \(<C, A.f(), B.f()> \in \text{LC-Add}\), we find all graph nodes which call \(A.f()\) and add a new callee node \(B.f()\) for each of them
  – For each triple \(<C, A.f(), B.f()> \in \text{LC-Del}\), we find all callers which call \(B.f()\) and delete their callee nodes \(A.f()\)

• **AIC** changes
  – Use tuple \(<\text{join point}, \text{advice}>\)
  – Find each affected join point, then add or delete shadow edges between join point and advice
Example (2)

class A{
    void n() {
    static void fun(A a){
        a.n();
    }
}
class B {
    void n() {
}
}  
aspect M{
    before(): call(* B.n()){
    }
}

A.fun
A.n
B.n
M. before

class A{
    void n() {
    static void fun(A a){
        a.n();
    }
}
class B extends A{
    void n() {
}
}  
aspect M{
    before(): call(* B.n()){
    }
}

LC-Add(B, A.n, B.n)

AIC-Add(M.before, B.n)

Call site
**Overall Step**

- **Inputs:** an original call graph and a set of atomic changes
- **Outputs:** the updated call graph

**Diagram:**
- **Old CG**
- **Change set**
- **Classify atomic changes**
- **Update shadow edges**
- **Update virtual call sites**
- **Update edges for modified node**
- **Add new nodes**
- **Delete existing nodes**
- **CM, CAB, CIMB**
- **AM, INM, AEA**
- **DM, DIM, DEA**
- **AIC**
- **LC**
Empirical Evaluation

- 24 versions of eight AspectJ benchmarks collected from a variety of sources as our subject programs
  - abc benchmark package
  - ajc example package

<table>
<thead>
<tr>
<th>Programs</th>
<th>#Loc</th>
<th>#Ver</th>
<th>#Me</th>
<th>#Shad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>111</td>
<td>3</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>Figure</td>
<td>147</td>
<td>4</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>Bean</td>
<td>199</td>
<td>3</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>Tracing</td>
<td>1059</td>
<td>4</td>
<td>44</td>
<td>32</td>
</tr>
<tr>
<td>NullCheck</td>
<td>2991</td>
<td>5</td>
<td>196</td>
<td>146</td>
</tr>
<tr>
<td>Lod</td>
<td>3075</td>
<td>2</td>
<td>220</td>
<td>1103</td>
</tr>
<tr>
<td>Dcm</td>
<td>3423</td>
<td>2</td>
<td>249</td>
<td>359</td>
</tr>
<tr>
<td>Spacewar</td>
<td>3053</td>
<td>2</td>
<td>288</td>
<td>369</td>
</tr>
</tbody>
</table>
Empirical Evaluation

• Procedure
  – Take each successive version pair of a benchmark (e.g. v1 and v2, v2 and v3, etc) and the call graph of the initial version (i.e. the v1 version of v1, v2 pair) as the input

• Result
  – Count the number of the atomic changes and updated nodes/edges between two successive versions
  – Compare the effectiveness between exhaustive CHA algorithm and our incremental approach
## Updated Nodes and Edges

<table>
<thead>
<tr>
<th>Version</th>
<th>Nodes</th>
<th>Edges</th>
<th>Atomic Change</th>
<th>Related Change</th>
<th>Updated nodes%</th>
<th>Updated edges%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>12</td>
<td>18</td>
<td>23</td>
<td>14</td>
<td>63%</td>
<td>41%</td>
</tr>
<tr>
<td>Q3</td>
<td>12</td>
<td>18</td>
<td>38</td>
<td>22</td>
<td>67%</td>
<td>56%</td>
</tr>
<tr>
<td>F2</td>
<td>23</td>
<td>32</td>
<td>22</td>
<td>17</td>
<td>40%</td>
<td>59%</td>
</tr>
<tr>
<td>F3</td>
<td>25</td>
<td>31</td>
<td>80</td>
<td>62</td>
<td>81%</td>
<td>93%</td>
</tr>
<tr>
<td>F4</td>
<td>45</td>
<td>74</td>
<td>59</td>
<td>43</td>
<td>89%</td>
<td>60%</td>
</tr>
<tr>
<td>B2</td>
<td>21</td>
<td>37</td>
<td>35</td>
<td>23</td>
<td>53%</td>
<td>61%</td>
</tr>
<tr>
<td>B3</td>
<td>24</td>
<td>37</td>
<td>11</td>
<td>8</td>
<td>48%</td>
<td>40%</td>
</tr>
<tr>
<td>T2</td>
<td>36</td>
<td>90</td>
<td>41</td>
<td>27</td>
<td>74%</td>
<td>46%</td>
</tr>
<tr>
<td>T3</td>
<td>36</td>
<td>90</td>
<td>72</td>
<td>49</td>
<td>76%</td>
<td>71%</td>
</tr>
<tr>
<td>T4</td>
<td>36</td>
<td>86</td>
<td>37</td>
<td>32</td>
<td>48%</td>
<td>62%</td>
</tr>
<tr>
<td>N2</td>
<td>157</td>
<td>264</td>
<td>35</td>
<td>31</td>
<td>18%</td>
<td>19%</td>
</tr>
<tr>
<td>N3</td>
<td>157</td>
<td>259</td>
<td>7</td>
<td>6</td>
<td>15%</td>
<td>17%</td>
</tr>
<tr>
<td>N4</td>
<td>157</td>
<td>298</td>
<td>2</td>
<td>1</td>
<td>30%</td>
<td>15%</td>
</tr>
<tr>
<td>N5</td>
<td>157</td>
<td>247</td>
<td>2</td>
<td>1</td>
<td>14%</td>
<td>8%</td>
</tr>
<tr>
<td>L2</td>
<td>173</td>
<td>589</td>
<td>1979</td>
<td>1492</td>
<td>70%</td>
<td>77%</td>
</tr>
<tr>
<td>D2</td>
<td>183</td>
<td>598</td>
<td>85</td>
<td>58</td>
<td>44%</td>
<td>53%</td>
</tr>
<tr>
<td>S2</td>
<td>104</td>
<td>90</td>
<td>72</td>
<td>47</td>
<td>37%</td>
<td>65%</td>
</tr>
</tbody>
</table>

*more related atomic changes will lead to more node and edge updates in most cases*
## Construction Time

<table>
<thead>
<tr>
<th>Version</th>
<th>Exhaustive Time (ms)</th>
<th>Incremental Time%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>281</td>
<td>6%</td>
</tr>
<tr>
<td>Q3</td>
<td>104</td>
<td>14%</td>
</tr>
<tr>
<td>F2</td>
<td>275</td>
<td>46%</td>
</tr>
<tr>
<td>F3</td>
<td>94</td>
<td>33%</td>
</tr>
<tr>
<td>F4</td>
<td>333</td>
<td>15%</td>
</tr>
<tr>
<td>B2</td>
<td>297</td>
<td>11%</td>
</tr>
<tr>
<td>B3</td>
<td>168</td>
<td>9%</td>
</tr>
<tr>
<td>T2</td>
<td>276</td>
<td>5%</td>
</tr>
<tr>
<td>T3</td>
<td>104</td>
<td>45%</td>
</tr>
<tr>
<td>T4</td>
<td>551</td>
<td>6%</td>
</tr>
<tr>
<td>N2</td>
<td>521</td>
<td>33%</td>
</tr>
<tr>
<td>N3</td>
<td>357</td>
<td>13%</td>
</tr>
<tr>
<td>N4</td>
<td>625</td>
<td>10%</td>
</tr>
<tr>
<td>N5</td>
<td>484</td>
<td>10%</td>
</tr>
<tr>
<td>L2</td>
<td>1953</td>
<td>86%</td>
</tr>
<tr>
<td>D2</td>
<td>1156</td>
<td>19%</td>
</tr>
<tr>
<td>S2</td>
<td>74</td>
<td>37%</td>
</tr>
</tbody>
</table>

Note: does not include the time for generating atomic changes

- In most cases, our incremental algorithm can result in a great improvement (76% decrease on average)

Shown as a percentage compared with the exhaustive time.
Related Work

- **Atomic Changes Representation**
  - Change Impact Analysis for Java [Ren et al OOPSLA ‘04]
  - Change Impact Analysis for AspectJ [Sai et al ICSM ‘08]

- **Call Graph Construction**
  - CHA [Dean et al ‘95], RTA [Bacon et al ‘96], …
  - A framework for control flow analysis of AspectJ program [Si et al AOAsia ‘08]

- **Incremental analysis**
  - Incremental call graph reanalysis for object-oriented software maintenance [Souter et al ICSM ‘01]
Motivation and Background
  – Incremental analysis of call graph
  – AspectJ semantics and analysis challenges
  – Atomic changes and call graph for AspectJ

Our incremental approach
  – Atomic change classification
  – Relationship between atomic change and call graph
  – Incremental algorithm of constructing call graph for AspectJ
  – Experimental Evaluation

Conclusion and future work
Conclusion

- We presented an incremental call graph reanalysis algorithm for AspectJ software
  - Reuse the existing analysis result *atomic change*, which represents the semantic differences between two program versions
  - Explore the relationship between atomic changes and AspectJ call graph, and update the initial constructed call graph incrementally

- An average 76% decrease in the call graph reanalysis cost, compared with exhaustive analysis
Future directions

- Apply the basic idea of our reanalysis approach to other call graph?
  - e.g. RTA, CTA, etc.
  - How to preserve the precision

- Use atomic changes to construct control flow graph (CFG) or system dependence graph (SDG) incrementally?
  - Atomic changes are method-level representation
  - Lack information that CFG or SDG requires
Thanks!

Questions?