Shanghai Jiao Tong University
Software Theory and Practice Group

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

SJTU.EDU.CN

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:**

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

```
class C {
  void n(){...}
}               Join point
```

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:**

```
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

Abbreviation	Atomic Change Name
AA	Add an Empty Aspect
DA	Delete an Empty Aspect
INF	Introduce a New Field
DIF	Delete an Introduced Field
CIFI	Change an Introduced Field Initializer
INM	Introduce a New Method
DIM	Delete an Introduced Method
CIMB	Change an Introduced Method Body
AEA	Add an Empty Advice
DAP	Delete an Aspect Precedence
ASED	Add a Soften Exception Declaration
DSED	Delete a Soften Exception Declaration
AIC	Advice Invocation Change

Table 1: A catalog of atomic changes in AspectJ

ctJ

nformation, which is

*Reflects the semantic different between the original program **P** and edited program **P'**, in forms of <joinpoint, advice> matching tuples*

The formal definition of **AIC** is shown as follows:

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

AIC | Advice Invocation Change

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**)**

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

Example: Syntactic 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)

CAB depends on **AEA** => **CAB** $\prec$ **AEA**

AEA depends on **ANP** => **AEA** $\prec$ **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** $\prec$ **AEA**

Change Impact Analysis Model

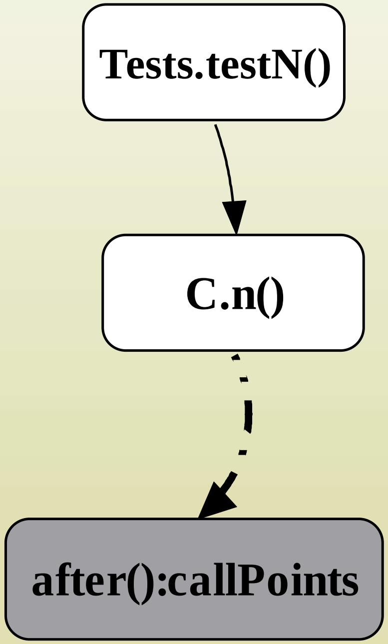
- **A change impact analysis model for AspectJ programs**
  - Used to identify affected program fragments, affected regression tests, and their corresponding changes

- **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, joinpont> as edges
  - Finally connect base code and aspect code graph
    - Conservative assumption for dynamic pointcut

Example: call graph

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

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

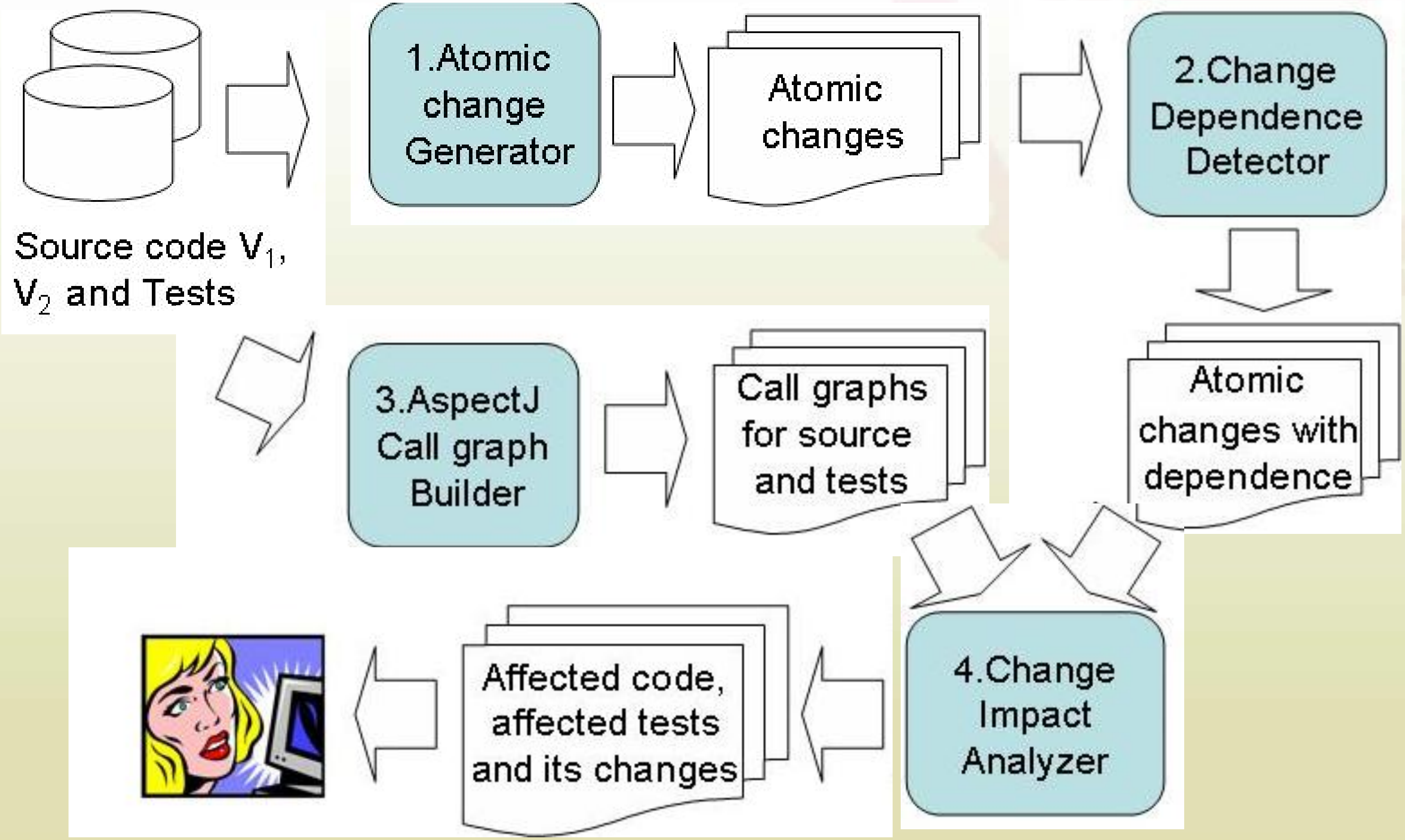


Impact Analysis Model

- 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

Tool Implementation

- 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

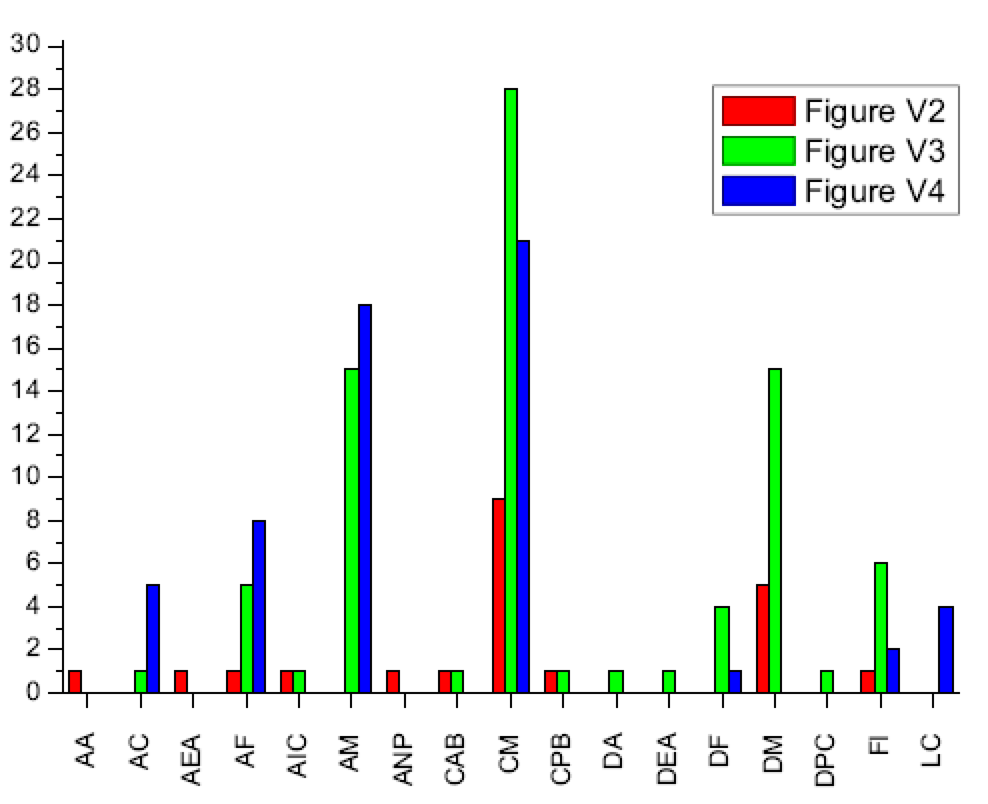
- **Subject programs**

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

SJTU.EDU.CN

Experimental Result (1)

- Atomic changes between version pairs



Celadon successfully handle aspectual features.

Experimental Result (2)

- Affected tests and affecting changes

Version	Total Number	% at	% ac
Q2	24	100%	67%
Q3	38	100%	71%
F2	22	60%	55%
F3	80	80%	58%
F4	59	30%	17%
B2	35	80.0%	86%
B3	11	40%	100%
T2	41	100%	
T3	69	100%	48%
T4	37	100%	73%
N2	35	78%	89%
N3	7	78%	86%
N4	2	51%	100%
L2	1979	100%	75%
D2	85	86%	67%
S2	74	30%	85%

Faulty change isolation

For regression test selection

Experimental Result (3)

- Affected program fragment (at method level)

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

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, Misherghi 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 `cflow`
- **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.