Change Impact Analysis for AspectJ Programs

Sai Zhang, Zhongxian Gu, Yu Lin, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{saizhang, ausgoo, linyu1986, zhao-jj}@sjtu.edu.cn

Abstract

Change impact analysis is a useful technique for software evolution. It determines the effects of a source editing session and provides valuable feedbacks to the programmers for making correct decisions. Recently, many techniques have been proposed to support change impact analysis of procedural or object-oriented software, but seldom effort has been made for aspect-oriented software. In this paper we propose a new change impact analysis technique for AspectJ programs. At the core of our approach is the atomic change representation which captures the semantic differences between two versions of an AspectJ program. We also present an impact analysis model, based on AspectJ call graph construction, to determine the affected program fragments, affected tests and their responsible changes. The proposed techniques have been implemented in Celadon, a change impact analysis framework for AspectJ programs. We performed an empirical evaluation on 24 versions of eight AspectJ benchmarks. The result shows that our proposed technique can effectively perform change impact analysis and provide valuable information in AspectJ software evolution.

1 Introduction

During software evolution, software change is an essential operation that introduces new functionalities or fixes bugs in the existing system, or modifies the former implementation if the requirements were not correctly addressed. After a long session of code editing, the nontrivial combination of small changes may affect other parts of the program unexpectedly, such as the non-local effect in object-oriented languages due to the extensive use of sub-typing and dynamic dispatch. Those likely ripple-effects of software change may lead to potential defects in the updated version. When regression tests fail, it may be difficult for programmers to locate the faulty changes by searching through the source. Moreover, when programmers develop regression tests over time to confirm the modified functionalities, it is also difficult to have a clear view of whether the existing tests are adequate to cover all affected parts.

Software change impact analysis [4] is a technique that assesses which parts of a program can be affected if a proposed change is made. It can be used to predict the potential impact of changes before they are applied, or to estimate the side-effect of changes after they are addressed. The information provided by change impact analysis would be very important for developers to make correct decisions.

Aspect-Oriented Programming (AOP) [9] has been proposed as a technique for improving separation of concerns in software design and implementation. An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the program. With the inclusion of join points, an aspect woven into the base code is solely responsible for a particular crosscutting concern, which raises the system's modularity. However, when implementing changes in AspectJ programs, it involves more complex impacts than in the traditional programming languages, such as: **(1)** the woven aspect may change control and data dependency of the base code. When either changes in base code (like renaming classes, fields and methods) or aspect code (like adding a new pointcut or advice) occur, the program behavior may be silently altered and no compiler or weaver can guarantee the correctness; and **(2)** failures of regression tests could either result from changes in the base code or a particular aspect. The more complex cases result from the interactions between the base and aspect code or even the aspect weaving sequences. In such a case, no single location corresponds to the failure, and the difficulty of finding failure-inducing changes rises dramatically.

Although many change impact analysis techniques have been presented in the literature, most of the work has been focused on procedural or object-oriented software [3, 4, 10, 12, 13]. Since the executable code of an AspectJ program is pure Java bytecode, an obvious approach is to apply di-

rectly the existing techniques for Java to the bytecode. In fact, this requires to build and maintain a map that associates the effects of each element in the bytecode back to those of its corresponding element in the source code. However, as pointed out in Xu's experimental study [17], there is a significant discrepancy between the AspectJ source code and the woven Java bytecode. This makes it extremely hard to establish such an association map. For example, the *cflow* pointcut requires code to be woven at many places in the base program, both at the so-called "update" shadows and the "query" shadows [8]. Moreover, the mapping relationship between source-code-level and byte-code-level is specific to the AspectJ compiler being used; different compilers or even different versions of the same compiler can create completely different mappings.

An alternative approach, which has been taken in this paper, is to perform change impact analysis on the source code of AspectJ programs. However, source code level analysis is complicated by the semantic complexity of the AspectJ programming language, such as pointcut types, multiple advices invoked at one join point, and the existence of dynamic advice. Therefore, an appropriate analysis approach which can handle the unique aspectual features in AspectJ programs is needed. In this paper, we propose a new change impact analysis technique for AspectJ programs. At the core of our approach is the *atomic change* representation, which captures the semantic differences between two AspectJ program versions. We construct aspect-aware call graphs [11] as the basis of our analysis. Using the atomic change and call graph representation, the affected program fragments, affected tests, and the responsible changes for each affected test can be identified.

The main contributions of this paper are threefold. **First**, we propose a new change impact analysis technique for AspectJ programs, with a catalog of *atomic changes* and their inter-relationships. **Second,** we present an impact analysis model for AspectJ programs, which handles the unique aspectual features and can be used to determine the affected program fragments, affected tests and their responsible changes. **Third,** we implemented Celadon, a change impact analysis framework for AspectJ programs; and performed an experimental evaluation of our proposed technique. The results show our approach is a promising solution to analyze the change impacts of AspectJ programs.

The rest of this paper is organized as follows. Section 2 introduces a motivating example that gives intuition about our approach. Section 3 presents a catalog of atomic changes for aspect-related constructs, with their inter-relationships. Section 4 presents a change impact analysis model for AspectJ programs. In Section 5, we describe the tool implementation and present an empirical evaluation of our technique. Related work and concluding remarks are given in Section 6 and Section 7, respectively.

2 Motivating Example

We use a small example shown in Figure 1 to give an informal overview of our approach. Associated with the program are three JUnit tests, `testA`, `testB` and `testC` as shown in Figure 1. The original version of the program consists of all program fragments except for those marked by underline. Here, we assume the editing parts to the original program are all new added and marked by underline.

```
class A {
  public int i, j;
  public void m() {i++;}
  public void n() {j++;}
}
class B extends A { }
class C extends B {
  public void n(){if(i > 0)  j = j + 2; m();}
}
aspect M {
  public void B.m() { i = i+2;}
  pointcut callPoints(A a):
    execution(* C.n()) && this(a);
  after(A a): callPoints(a) { a.j ++;}
}
public class Tests extends TestCase {
    public void testA() {
        A a = new A(); a.m(); a.n();
        assertTrue(a.i == a.j);
    }
    public void testB() {
        A a = new B(); a.m(); a.n();
        assertTrue(a.i == a.j);
    }
    public void testC() {
        A a = new C(); a.m(); a.n();
        assertTrue(a.i == a.j);
    }
}
```

Figure 1. A sample program with JUnit tests.

Our approach first decomposes the source code editing into a set of atomic changes (Table 1 and Table 2). These atomic changes represent the source modifications at a coarse-grained model, and capture the semantic differences between the original and the edited program. Additionally, there are syntactic dependencies between atomic changes. That is, an atomic change C_1 is dependent on another atomic change C_2 (denoted as, $C_2 \preceq C_1$), if applying C_1 to the original version of the program without also applying C_2 will cause a syntactically invalid program that contains some, but not all of the atomic changes.

Example: Figure 2 shows the atomic changes with their dependence relationships inferred from the two versions of the sample program. Each atomic change is shown as a box, where the top half of the box shows the category of the change, and the bottom half shows the method, field or advice involved. An arrow from an atomic change C_1 to C_2 indicates that C_2 is dependent on C_1.

Besides the atomic change representations, another core part of our approach is the call graph representation for AspectJ programs. A call graph is constructed to determine: (1) the affected code fragments, (2) the affected tests after

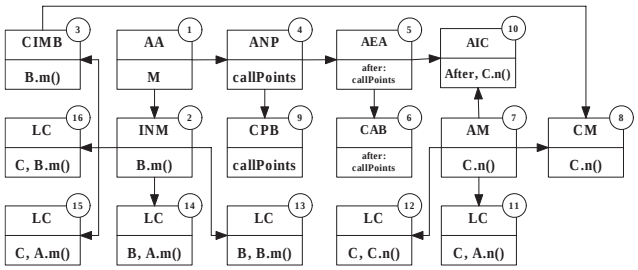

Figure 2. Atomic changes inferred from the sample program, with their dependencies.

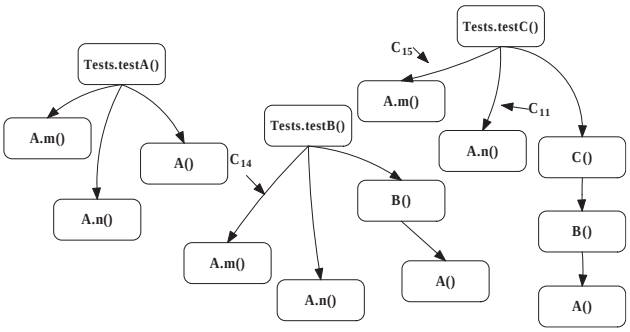

Figure 3. Call graphs for the tests in the original program version.

program changes; and (3) the affecting atomic changes for each affected test.

Figure 3 shows the dynamic call graphs [1] for three tests `testA`, `testB`, and `testC`. A test is determined to be affected if: (1) its call graph contains a node that corresponds to a base code change [12], like a changed method (**CM**), or contains an edge that corresponds to a lookup change **LC**; and (2) its call graph contains a node that corresponds to a **CAB**, **DEA**, **CIMB**, or **DIM** change or contains an edge that corresponds to an advice invocation change **AIC** or contains a node involved in an **AIC** change. Using the call graphs in Figure 3, we can see that `testA` is not affected, while `testB` and `testC` are affected.

For the affected tests (`testB` and `testC`), their call graphs on the updated version are shown in Figure 4. We use shadows to annotate the modified method or advice, and edges corresponding to advice invocation are labelled with dashed lines. The set of *affecting changes* that affect a given test t_i includes: (1) atomic changes occurred in the base code, changes like changed methods (**CM**) and added methods (**AM**) that correspond to a node in the edited call graph, and changes like lookup change (**LC**) that correspond to an edge in the call graph, and (2) the atomic changes appearing in the aspect code, including changes of adding new advice (**AEA**), changing advice body (**CAB**), introducing a new inter-type declared method (**INM**) and changing an inter-typed declared method body (**CIMB**) that correspond to a node in the call graph. The affecting atomic changes also include the advice invocation changes (**AIC)** that correspond to an edge in the call graph. The whole affecting atomic change set also includes the transitively prerequisite atomic changes of all above changes.

[1] The Celadon tool constructs static AspectJ call graphs for safely identifying impacted parts, and it can also work with call graphs constructed from the actual program execution.

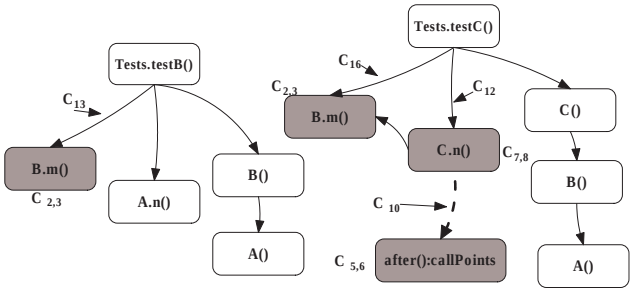

Figure 4. Call graphs for the affected tests in the edited program version.

Based on the dependence relationship defined in the call graph, the potentially affected fragments in the edited program, are all nodes that reach to the modified (or added) nodes, that is, the transitive closure of all modified nodes. The formal equations to compute affected tests, affecting changes and affected fragments are given in Section 4.

Example: There are 16 atomic changes inferred from the sample example as shown in Figure 2. Only test `testB` and `testC` are affected. For test `testB`, the affecting changes are change 1, 2, 3 and 13. And for test `testC`, the affecting changes are change 1, 2, 3, 4, 5, 6, 7, 8, 10, 12 and 16. In the edited program version, the affected code fragments are method `B.m()`, `C.n()` and advice `after: callPoints()`. Basically, this information provided by Celadon informs programmers that those affected methods and advices should be covered in the regression test suite.

3 Atomic Changes

We next present a catalog of atomic changes for AspectJ programs. These atomic changes focus on the aspect-related constructs and represent the source code modifications at a coarse-grained model (that is, method-level). We assume that the original and the updated programs are both syntactically correct and compilable, and any source code changes can be decomposed into a set of *atomic changes* that is amenable to analyze. Since AspectJ is a seamless extension to Java, the atomic changes identified for Java [13] listed in Table 2 should also be used to modeling changes in the base code of AspectJ programs.

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
DEA	Delete an Empty Advice
CAB	Change an Advice Body
ANP	Add a New Pointcut
CPB	Change a Pointcut Body
DPC	Delete a Pointcut
AHD	Add a Hierarchy Declaration
DHD	Delete a Hierarchy Declaration
AAP	Add an Aspect Precedence
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

3.1 Atomic Changes for AspectJ

Table 1 shows a catalog of atomic changes we defined for AspectJ programs. Most of these changes are self-explanatory except for **AIC**, which will be explained later in details. We ignore several types of source code changes that have no direct semantic impact on the program behavior, such as changes to access right of aspects, pointcuts, inter-type declarations, and the addition or deletion of a **declare error** statement, which only affects compile-time behavior.

3.1.1 Aspect, Pointcut, and Advice Changes

Seven atomic changes in Table 1 correspond to the changes of aspect, pointcut and advice constructs. **AA** and **DA** denote the set of adding and deleting an empty aspect, respectively. Similarly, **ANP** and **DPC** denote the adding and deleting of a pointcut, while **CPB** denotes the change of a pointcut body. **AEA**, **DEA** and **CAB** capture the changes of adding, deleting and modifying advice body, respectively. Note that adding an advice, for example, the advice `after(A a):callPoints(a)` of aspect `M` in Figure 1, is decomposed into three steps: the addition of pointcut `callPoints` (**ANP** change), the addition of an empty advice `after(A a):callPoints(a)` (**AEA** change), and the insertion of advice body (**CAB** change). Each step is mapped to one atomic change.

3.1.2 Inter-Type Declaration Changes

Six atomic changes in Table 1 represent the changes caused by the inter-type declaration mechanism in AspectJ. **INF**, **DIF** and **CIFI** denote the set of adding, deleting and changing of an inter-type field. **INM**, **DIM** and **CIMB** capture changes of introducing a new method, deleting, and changing an existing method, respectively. For example, the introduction of method `B.m()` in Figure 1 is decomposed into two steps: introducing a new method declaration and changing the method body, which correspond to atomic changes **INM** and **CIMB**, respectively.

3.1.3 Hierarchy and Aspect Precedence Changes

Changes to the class hierarchy or the aspect weaving precedence also have significant effects on program behavior. We use two atomic changes **AHD** and **DHD** to denote the actions of adding and deleting a hierarchy declaration in AspectJ. Another two atomic changes **AAP** and **DAP** are used to capture the aspect precedence changes. For simplicity, in our approach, any changes to the hierarchy declaration (such as the **declare parent** statement) or aspect precedence declaration will be transformed into two steps: deleting the existing declaration and adding a new one.

3.1.4 Soften Exception Changes

AspectJ provides a special mechanism to specify that a particular kind of exception, if thrown at a join point, should bypass Java's usual static exception checking system and instead be thrown as an `org.aspectj.lang.SoftException`. We define two changes **ASED** and **DSED** to capture adding and deleting soften exception declarations. Any change to the soften exception declaration is decomposed into two steps: deleting the original declaration and adding a new one.

3.1.5 Advice Invocation Changes

Changes to the base or aspect code may cause lost or additional matches of join points, which may result in accidental advice invocations. Therefore, we define the atomic change **AIC** to represent the advice invocation change. The **AIC** change reflects the semantic differences between an original version *P* and an edited program version *P'* in the form of a set of tuples $< joinpoint, advice >$, which indicate that the *advice* invoking at the above *join point* has been changed. The formal definition of **AIC** is shown as follows:

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

where J and A are the sets of join points and advices in the original program, and J' and A' are the sets of join points and advices in the modified program. $J \times A$ denotes the matched join point and advice tuple set in the original program, while $J' \times A'$ denotes the matched tuple set in the updated program version.

3.2 Inter-relationships between Atomic Changes

There are some inter-relationships between atomic changes that induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. That is, C1 $\preceq^*$ C2 denotes that

Abbreviation	Atomic Change Name
AF	Add a field
DF	Delete a Field
FI	Field Initialization
AM	Add an Empty Method
DM	Delete an Empty Method
CM	Change Body of Method
AC	Add an Empty Class
DC	Delete an Empty Class
LC	Change Virtual Method Lookup

Table 2. Atomic changes in base code

C1 is a prerequisite for C2 [2]. This partial ordering indicates that when applying one atomic change to the program, all its dependent changes should also be applied in order to obtain a syntactically valid version. The benefit of defining such a partial ordering is that one can construct a semantic correctness intermediate version for future analysis [6, 19]. We define two types of dependencies as follows.

3.2.1 Syntactic Dependence

Generally, syntactic dependence captures all prerequisite changes for one atomic change in order to form a valid version. The *syntactic dependencies* between atomic changes of base code have been discussed in [12]. Here we just focus on the aspect-related constructs.

Adding / Deleting AspectJ Constructs. In our definition, all the adding atomic changes (**AA, INF, INM, AEA** and **ANP**) and deleting atomic changes (**DA, DIF, DIM, DEA** and **DPC**) represent adding or deleting an empty AspectJ construct. A new construct must be declared before using or making changes to its body, and similarly the construct body must be cleared before deleting the construct itself. For example, the new added advice `after(A a):callPoints(a)` in Figure 1 must be declared first before making changes in its body block, and the pointcut definition `callPoints(A a)` used in this advice must be added before declaring the advice, and all above changes must be applied after the addition of the empty aspect `M` . The syntactic dependencies between atomic changes involved in Figure 1 can be represented as:

AEA(`after(A a):callPoints(a)`) $\prec$
CAB(`after(A a):callPoints(a)`)
ANP(`callPoints`) $\prec$
AEA(`after(A a):callPoints(a)`)
AA(`M`) $\prec$ **ANP**(`callPoints(A a)`)

Similarly, the dependencies in deleting changes can be represented in a reverse ordering.

Changing AspectJ Constructs. In our model, only changes to the advice body, pointcut body, inter-type method body and inter-type field initializer have corresponding atomic changes (i.e., **CAB**, **CPB**,**CIMB** and **CIFI**). Other changes to the aspect constructs, such as *changing type of an inter-type field* or *renaming an inter-type method*, are decomposed into a delete change (**DIF**, **DIM**), an add change (**INF**, **INM**), and a corresponding modification change (**CIFI**, **CIMB** or **CAB**). In this case, the constructs must be declared first before making any changes.

Class Hierarchy and Aspect Precedence Changes. Change to the class hierarchy or aspect precedence declaration is related to the class or aspect definition it uses. An aspect must be defined before it is used in any aspect precedence statement, but deleting any aspect precedence does not need any prerequisites. For example, suppose we add a new aspect `N` and an aspect precedence declaration:

```
declare precedence M:N
```

as extra changes in Figure 1. This kind of dependence can be represented as:

AA(`M`)$\prec$**AAP**(`M,N`)
AA(`N`)$\prec$**AAP**(`M,N`)

The dependencies in class hierarchy changes can be handled in a similar manner: the class or interface used in the declaration must be declared first.

Soften Exception Changes. Changes to the soften exception declaration are related to the pointcut definition it uses. The constructs in pointcuts must be first defined before used in any soften exception declaration. For example, suppose that we add a soften exception declaration

```
declare soft: Exception:
    execution(C.n());
```

as an extra change in Figure 1, the dependence can be represented as:

AM(`C.n()`) $\prec$ **ASED**(`execution(A.p())`)

3.2.2 Interaction Dependence

The dependencies we discuss so far are all syntactic dependencies in AspectJ programs, while the *interaction dependence* is the implicit inter-relationship involving both atomic changes in the aspect code and base code.

Inter-type Declaration Changes. As discussed in the previous section, the inter-type declaration should be defined before making any changes. Take the changes in Figure 1 as an example, method `B.m()` must be first introduced by aspect `M` before it is used in method `C.n()`. Similarly, an introduced field can only be deleted when it is no longer referenced in base code. Hence, for the changes in Figure 1, we have the following dependencies related to the inter-type declaration change:

INM(`B.m()`) $\prec$ **CM**(`C.n()`)

Overriding Methods. When the inter-type method introduced by aspect code overrides the existing one in the base code, it will change the method dispatch and cause an **LC**

[2] To keep consistence, we use the same notions for the partial order definition as used in [13]

atomic change. This **LC** depends on the inter-type declaration introduced. Consider the change in Figure 1, aspect `M` introduces a method `m()` for class `B`. For this change, an object of type `C` will no longer resolve to `A.m()`. This method dispatch change can be represented by a **LC** change. Here, the dependence between the **LC** change and the inter-type declaration can be represented as:

INM(`B.m()`) $\prec$ **LC**

Abstract Aspects and Pointcuts. An abstract pointcut must be implemented in all the sub-aspects of an abstract aspect. Therefore, the declaration of an abstract pointcut must be deleted before the deletion of its implementation in the sub-aspect. This dependence relationship captures the way a new abstract pointcut is declared or deleted. Assume that program `P` defines an abstract aspect `AA` with a sub-aspect `SA`, which extends `AA`. We add an abstract pointcut declaration of `pcA` into `AA`, and `SA` provides the implementation of `pcA`. Here, we have the following dependence between atomic changes:

ANP(`SA.pcA()`) $\prec$ **ANP**(`AA.pcA()`)

Advice Invocation Changes. In AspectJ program, either changes like *renaming class members* in base code or changes like *adding new advice* in aspect code will cause the advice invocation changes. The advice invocation changes depend on the related source modifications. Consider the changes in Figure 1, the new added pointcut `callPoints` matches the new added method `C.n()`. The corresponding **AIC** is represented as **AIC** = {<`C.n()`, `after(A a):callPoints(a)`>} in our model, and the dependence relationship can be represented as:

AM(`C.n()`) $\prec$ **AIC**
AEA(`after(A a):callPoints(a)`) $\prec$ **AIC**

4 Change Impact Analysis Model

We next present a change impact analysis model for AspectJ programs. Our model relies on the construction of AspectJ call graphs, and aims to determine all affected program fragments, affected tests and their responsible changes. Let *P* be an AspectJ program and *Nodes(P)* and *Edges(P)* be the node and edge sets in the call graph of *P*. We also assume there is a set of tests $T = t_1, ..., t_n$, associated with the original program *P*. Each test driver t_i exercises a subset *Nodes*(P, t_i) of *P*'s method and a subset *Nodes*(P', t_i) from the call graph for t_i on the edited program P'.

To assist our analysis, we define the notation of *Affected Node Set ANS(S)* as all potential affected nodes in the call graph, where *S* is the modified node set. *ANS(S)* represents all nodes reachable from any node $s \in S$, that is, the transitive closure of *S* in the graph.

4.1 The Affected Program Fragments

Either changes in base code or aspect code can affect the behavior of AspectJ programs. The affected base code **AffectedBase1** due to changes of base code can be computed by the formalism defined in [13].The affected base code **AffectedBase2** caused by changes in aspect code is:

AffectedBase2 $\equiv$ ***ANS*** { **AffectedBaseNodes** }
AffectedBaseNodes$\equiv$ { *m* | *m* $\in$ *Nodes(P')* $\wedge$ <*m*, *a*> $\in$ **AIC** }

Thus, the total affected base code **AffectedBase** is:

AffectedBaseParts $\equiv$ **AffectedBase1** $\cup$ **AffectedBase2**

Similarly, the affected aspect code can be represented as:

AffectedAspectParts $\equiv$ ***ANS*** { **AffectedAspectNodes** }
AffectedAspectNodes $\equiv$
{ *n* | *n* $\in$ **INM** $\cup$ **DIM** $\cup$ **CIMB** $\cup$ **AEA** $\cup$ **CAB** $\cup$ **DEA** }
$\cup$ { *m* | <*m*, *a*> $\in$ **AIC** }

Combining both affected base code and aspect code fragments, all the affected fragments in an AspectJ program can be represented as:

AllAffectedNodes $\equiv$ **AffectedBaseParts** $\cup$ **AffectedAspectParts**

4.2 Affected Tests and Affecting Changes

The affected test cases are the subset of existing test cases whose behavior may be affected by the changes. A test is determined to be affected if its call graph contains an affected node.

The formal definition of *affected tests* shows as follows:

AffectedTests $\equiv$
{ t_i | t_i $\in$ T, **AllAffectedNodes** $\cap$ *Nodes(P',t)* $\neq \emptyset$ }

The set of *affecting changes* that affects a given test t_i includes the transitively prerequisite of all affected atomic changes appearing on its call graph.

The formal definition of *affecting changes* shows as below. Here ***A*** is the set of all atomic changes.

AffectingChanges $\equiv$
{ c' | $c \in$ *Nodes(P',t)* $\cap$ **AllAffectedNodes** $\cap$ ***A***, $c' \preceq^* c$}

5 Empirical Evaluation

To investigate the effectiveness and efficiency of our proposed technique, we have implemented Celadon, a change impact analysis framework for AspectJ programs. Celadon is designed as an Eclipse plugin and built on top of the *ajc* AspectJ compiler [2]. It uses a dynamic programming algorithm [18] to compare the abstract syntax tree (AST) of two program versions, generating the corresponding atomic change set. When constructing AspectJ call graphs, Celadon uses the RTA algorithm [5] to build the call

graph of the base code. We consider the advice as a method-like node with matching relationship represented by an edge from the join point. The complete AspectJ call graph is formed after the call graph of aspect code is *connected* into the base code call graph using the join point matching information. For more implementation details, please refer to [20].

5.1 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

Table 3. Subject Programs

We evaluated Celadon on 24 versions of eight AspectJ benchmarks collected from a variety of sources. The first three and the spacewar example are included in the AspectJ compiler example package. Other programs are obtained from the abc benchmark package [1]. This group of benchmarks has also been widely used by other researchers to evaluate their work [16] [7] [17]. For each program, we made the first version v_1 a pure Java program by removing all aspectual constructs. We also developed a high coverage test suite for each benchmark.

Table 3 shows the number of lines of code in the original program (#Loc), the number of versions (#Ver), the number of methods (#Me), the number of shadows (#Shad), the size of the test suite (#Tests), the percentage of methods covered by the test suite (%mc), and the percentage of advice shadows covered by the test suite (%asc).[3]

We take each pair of successive versions of an AspectJ benchmark (i.e., v_1 and v_2, v_2 and v_3) and its corresponding tests as the input of our tool. The tests are used in both versions. For each input, change impact analysis is performed automatically by Celadon and the result is reported in terms of affected program fragments, affected tests and its responsible affecting changes.

5.2 Results

Figure 5 shows the number of atomic changes between version pairs of each benchmark. All atomic changes are classified by categories, and the number varies greatly between 1 and 676. Table 4 summarizes the affected tests and affecting changes for each version pair of those benchmarks, and Table 5 shows the affected method nodes in call graph of the analyzed programs. In Figure 5, Table 4, and Table 5, each AspectJ program version is labelled with its number - eg., Q2 corresponds to version v_2 of Quicksort, N4 is version v_4 of Nullcheck. The experimental data between version v_{n-1} and v_n is shown in bar v_n or row v_n.

5.2.1 Atomic Changes

Quicksort: There are ten atomic change categories derived from the source modifications between two version pairs. As seen in the diagram, the most frequent changes are **AIC** (*Advice Invocation Change*) and **CAB** (*Change Advice Body*). They both have a number of eight between version v_2 and v_3. The second frequent change is **CPB** (*Change Pointcut Body*) , which accounts for seven between v_2 and v_3. Program version v_2 and v_3 of Quicksort have different pointcuts and advices to implement the crosscutting concerns. Thus, it is reflected by a number of corresponding atomic changes, such as **AEA** (*Add Empty Advice*), **DEA** (*Delete Empty Advice*), and **ANP** (*Add New Pointcut*).

Figure: Most of atomic changes in the Figure benchmark are in low number except for common Java changes like **CM**, **AM**, and **DM**. Version v_2 adds an aspect `DisplayUpdating`, which contains a pointcut `move()` together with an advice `after returning: move()`. This change is captured by one **AA**, one **ANP**, one **CPB**, one **AEA**, one **CAB**, and one **AIC** change.

Bean: There are 18 atomic change categories between the successive version pairs of the Bean benchmark, among which 13 are aspect-related. Version v_2 declares five inter-type methods (eg. method `Point.addPropertyChangeListener`) , which correspond to five **INM** (*Introduced New Method*) and five **CIMB** (*Change Introduced Method Body*) changes. Version v_2 adds a statement `declare parents:Point implements Serializable` in aspect `PointBound`, which is decomposed into an **AHD** (*Add Hierarchy Declaration*) change. The changes between v_2 and v_3 mainly come from the different implementation of pointcut and advice in the `BoundPoint` aspect. These changes are captured by three **AIC**, two **AEA**, and three **CAB** changes.

Tracing: The most frequent aspect-related change is **AIC**, while the overall most frequent change is the **CM**. There are four **AIC** changes between version v_1 and v_2 caused by additions of *new advices* and *new methods* (captured by four **AEA** changes). These **AIC** changes indicate the addition of advice, join point matching pairs in the updated program.

NullCheck: Though Nullcheck is a moderate-sized AspectJ application, there are not many changes between its successive versions. The number of **CM** changes is surprisingly higher than others between v_1 and v_2, which indicates there is an intensive editing of *changing the existing method body*. The source editing of aspect-related feature is mainly in the pack-

[3] *Advice shadow coverage* is defined as follows. An *Advice shadow interaction* occurs if a test executes an advice whose pointcut statically matches a shadow. The *Advice shadow coverage* is the ratio between *Advice shadow interaction*s and the number of shadows in program.

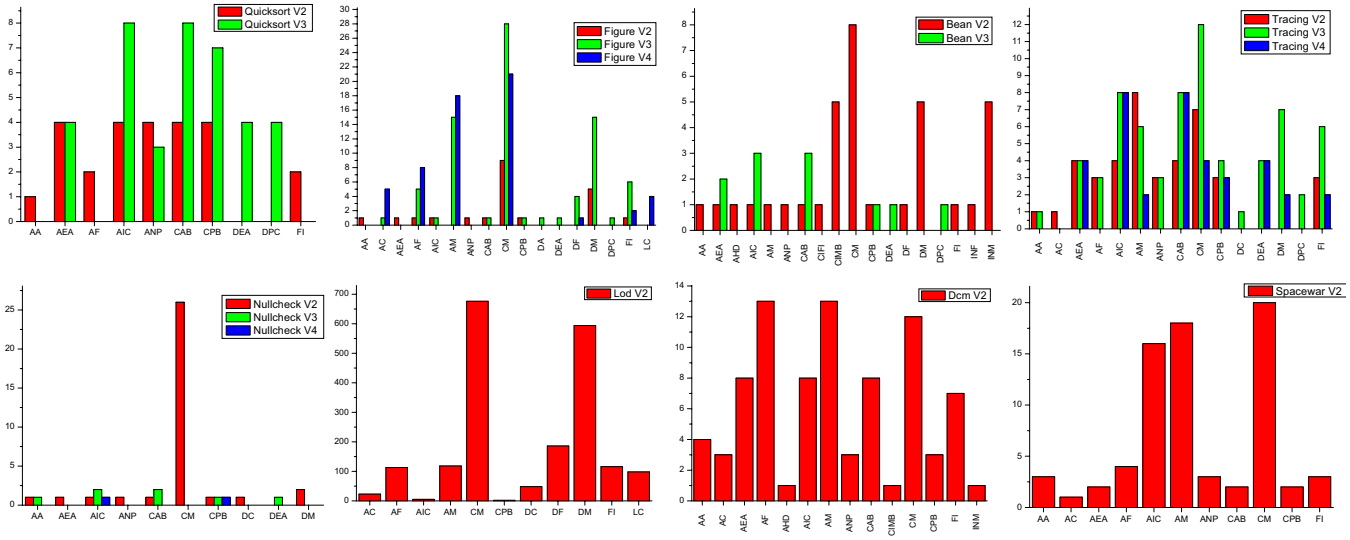

Figure 5. Number of atomic changes between version pairs of each subject program

age `lib.aspects.codingstandards`. For example, version v_3 changes an advice declaration `after() returning(Object lRetVal)` to `Object around(): methodsThatReturnObjects()`. Interestingly, there are only two atomic changes reported by Celadon between v_3 and v_4. These changes come from the modification of pointcut `methodsThatReturnObjects()` in aspect `EnforceCodingStandards`. This pointcut body change causes the loss of join point matching. Therefore, one **AIC** is generated to reflect the subtle semantic difference.

LoD: The number of changes is surprisingly high between two available versions. It varies greatly from 2 to 676, because there is a fundamental different package organization of source code between two versions. Version v_2 removed a top level package `weka` and added two new packages `jsim` and `certrevsim` in the *src* directory. These source changes under package `weka`, `jsim` and `certrevsim` are decomposed into 23 **AC**, 113 **AF**, 118 **AM**, 676 **CM**, 98 **DM**, and the corresponding deleting changes.

Dcm: Version v_2 adds a new package `DCM` to the original program. Four new added aspects `Pointcuts`, `ClassRelationship`, `AllocFree` and `Metrics` in package `DCM`, `DCM.handleGC`, and `DCM.handleMetrics` and also their member changes are captured by four **AA**, eight **AEA**, one **AHD**, eight **AIC**, three **ANP**, eight **CAB**, one **CIMB**, one **INM**, and three **CPB** changes.

Spacewar: Besides common Java atomic changes like **CM** and **AM**, the most frequent change is the **AIC** (*Advice Invocation Change*). These **AIC** changes indicate that the new added aspect has crosscutted the original program and may affect the program runtime behavior. For example, the new added aspect `RegistrationProtection` in the `Registry.java` file, which intercepts the calling to method `Registry.regiser(SpaceObject)` and `Registry.unregiser(SpaceObject)`, is used to check the designed invariant during the runtime.

5.2.2 Affected Tests and Affecting Changes

Table 4 shows the affected tests and their affecting changes for each benchmark version pair. On average, 74.2% of the tests are affected and 69.2% of the atomic changes are responsible in each version pair. Interestingly, there are several version pairs over which all tests are affected. For example, all tests are affected between version v_1 and v_2 (row T_2 in Table 4), despite the fact that there are only 41 atomic changes during this time. The reason is that version T_2 defines two pointcuts `myConstructor():myClass() && execution(new(..))` and `myMethod():myClass() && execution(* *(..))` in aspect `TraceMyClasses`, which crosscut all constructors and methods that are always executed at run time. It indicates that in AspectJ programs, even small changes (such as *adding new pointcut*) would dramatically change the program structure and affect its runtime behavior. Therefore, in principle, a change impact analysis tool could inform the user that all tests should be rerun to validate the changes when an observation of this kind is found.

The number in column %ac (affecting changes) indicates the percentage of responsible changes for all affected tests. For some versions, the affecting atomic changes percentage reported by Celadon is reduced to 16.9% (F_4), 36.5% (T_2) and 47.8% (T_3). This means that our approach has the potential application of isolating the responsible changes for enhancing program understanding and debugging [6, 19], when regression tests fail.

5.2.3 Affected Program Fragments in Call Graph

The percentage of affected program fragments (call graph nodes) are shown in Table 5, ranging from 10.6% to 95.6%. The data in row Q_2, N_2, and N_3 indicates that both *adding aspect features* into the original object-oriented systems and *changing the existing aspect features* would dramatically affect the whole program. In row N_2, nearly all of the nodes in the call graph are affected, because N_2 adds

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%	36%
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%

Table 4. Total atomic change number (Total Number), percentage of affected tests (% at), and percentage of affecting atomic changes (% ac)

a dynamic pointcut `methodsThatReturnObjects(): execution(Object+ *.*(..))` to cross cut base Java methods, which are always matched at runtime. Celadon reports such potentially affected methods in the analysis, and informs developers that sufficient tests should be developed to cover all these affected fragments.

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%

Table 5. Total method nodes in call graph (Nodes Num), number of affected method nodes (Affected Nodes) and percentage of affected nodes (% Affected Nodes)

5.3 Other Experimental Issues

Threat to Validity. Like any empirical evaluation, this study also has limitations which must be considered. Although we have applied Celadon on 24 versions of eight AspectJ benchmarks, which are well known examples and the last three ones are among the largest programs, they are smaller than traditional Java software systems. For this case, we can not claim that these experiment results can be necessarily generalized to other programs. On the other hand, threats to internal validity maybe mostly lie with possible errors in our tool implementation and the measure of experiment result. To reduce these kind of threats, we have performed several careful checks. For each result produced by Celadon, we manually inspected the corresponding code to ensure the correctness.

Analysis Cost. The analysis performed by Celadon runs in practical time. Our experiment is conducted on a DELL C521 PC with AMD Sempron 3.0 Ghz CPU and 1.0 GB memory. For the two largest programs `LoD` and `Dcm`, the total running time (including compilation time) of our analysis is 10.6 and 12.9 seconds on average, respectively.

6 Related Work

We next discuss some related work in the areas of change impact analysis and regression test selection techniques.

Change impact analysis. Many change impact analysis techniques [3, 4, 10, 12, 13] have been proposed in recent years, which are mainly focused on procedural or object-oriented languages. Ryder et al. [13] first uses the atomic changes to determine the effects of a session of source changes for Java programs. They give a formal definition of the dependencies between atomic changes and the corresponding tool support in [6] and [12]. Our work is an extension of the concept of atomic changes to aspect-related constructs to handle the intricacy of AspectJ programs.

Zhao [21] presents an approach to support change impact analysis of AspectJ software based on program slicing. Stoerzer and Graf [15] focus on the semantic modification of base code and introduce a delta analysis to handle the *fragile pointcut* problem, based on a comparison of the sets of matched join points for two program versions. Shinomi and Tamai [14] discuss the impact analysis of aspect weaving and its propagation through the base program and aspects. They focus on the change impact caused by the weaving aspect. In our work, we present a systematic catalog of atomic changes for AspectJ programs to capture program semantic changes in the source code level. The analysis model can also be used to determine the affected program fragments, affected tests and their responsible changes.

Regression test selection. Change impact analysis is also related to the regression test selection techniques for aspect-oriented programs [17, 22]. The regression test selection techniques aim at reusing tests from an existing suite to retest the new program version in order to reduce the testing effort. Our change impact analysis technique uses (or selects) affected tests to indicate to programmers that the test behavior has been affected and they need to be rerun.

Zhao [22] proposes a fine-grained regression test se-

lection technique for AspectJ programs based on program slicing, with the goal of computing the effects of program changes. In comparison to our work, Zhao uses more costly system dependence graph (SDG) construction and considers program changes at a statement level of granularity. Xu [17] also presents a safe regression test selection technique for AspectJ software. Like Zhao's work, Xu's approach relies on simultaneous traversal of two program representations (AspectJ control flow graphs) to identify the "dangerous" program elements representing differences in program behavior. However, our approach to find affected tests does not rely on a simultaneous traversal of two representations of the program to find semantic differences. Instead, our approach determines affected tests by computing atomic changes from two version of programs and constructing AspectJ call graphs. Therefore, it does not need information about test execution on both versions of the program. Investigating the cost/precision tradeoffs between these approaches is a topic for further research.

7 Concluding Remarks

In this paper we present a change impact analysis approach for AspectJ programs. Our approach uses an *atomic changes* representation to capture the semantic differences between AspectJ program versions. We construct AspectJ call graphs to effectively identify the impacted program fragments, affected tests and their responsible affecting changes. Our approach is based on the static analysis of the source code of AspectJ programs, therefore it abstracts away the low-level details that are specific to a compiler implementation. Our empirical study indicates the proposed technique can provides useful tool supports and valuable information during the software evolution.

As our future work, we will improve our current change impact model to capture the semantic changes more precisely, especially the dynamic pointcut changes. We also will also investigate more applications of this analysis approach, such as automatic debugging [19], incremental analysis and regression tests prioritization.

Acknowledgements This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Si Huang, Chong Shu, Cheng Zhang and Martin Georg for their insightful comments.

References

[1] The abc Compiler. `http://abc.comlab.ox.ac.uk/`.

[2] The AspectJ Team. The AspectJ Programming Guide. Online manual, 2003.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. 9th ESEC and 11th ACM SIGSOFT FSE*, pages 128–137.

[4] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[5] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *In OOPSLA '96 Conference Proceedings, San Jose, CA, October 1996*, pages 324–341, 2004.

[6] O. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for Java programs. In *Proc. International Conference on Software Maintenance (ICSM'2005)*, Budapest, Hungary, September 27–29, 2005.

[7] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of AspectJ programs. In *Proc. OOPSLA 2004* ., pages 150–169, 2004.

[8] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[10] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[11] N. Li. The call graph construction for aspect-oriented programs. Master's thesis, School of Software, Shanghai Jiao Tong University, March 2007 (in Chinese).

[12] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[13] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE'01*, pages 46–53, 2001.

[14] H. Shinomi and T. Tamai. Impact analysis of weaving in aspect-oriented programming. In *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 657–660, 2005.

[15] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proc. 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[16] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM Press.

[17] G. Xu and R. Atanas. Regression test selection for AspectJ software. In *In Proc. of the 29th International Conference on Software Engineering(ICSE07)*, pages 65–74, May 2007.

[18] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[19] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. AutoFlow: An automatic debugging framework for AspectJ programs. Technical Report SJTU-CSE-TR-08-01, Center for Software Engineering, SJTU, Jan 2008.

[20] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Celadon: A change impact analysis tool for Aspect-Oriented programs. In *Proc. 30th International Conference on Software Engineering (ICSE 2008 Companion)*, May 2008.

[21] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Proc. 5th International Workshop on Principles of Software Evolution*, pages 108–112, May 2002.

[22] J. Zhao, T. Xie, and N. Li. Towards regression test selection for AspectJ programs. In *Proc. 2nd workshop on Testing aspect-oriented programs*, pages 21–26, July 2006.