Incremental Call Graph Reanalysis for AspectJ Software

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

Abstract

Program call graph representation can be used to support many tasks in compiler optimization, program comprehension, and software maintenance. During software evolution, the call graph needs to remain fairly precise and be updated quickly in response to software changes. In this paper, we present an approach to incremental update, instead of exhaustive analysis of the initially constructed call graph in AspectJ software. Our approach first decomposes the source code edits between the updated and initial software versions into a set of atomic change representations, which capture the semantic differences. Then, we explore the relationship between atomic changes and call graph to incrementally update the initially constructed graph, instead of rebuilding it from the ground up. We implement the reanalysis approach on top of the ajc AspectJ compiler and perform an empirical study on 24 versions of eight AspectJ benchmarks. The experiment result shows that our approach can reduce a large portion of unnecessary reanalysis cost as program changes occur, and significant savings are observed for the incremental reconstruction of AspectJ call graph in comparison with an exhaustive analysis, with no loss in precision.

1. Introduction

Call graph construction is a key task required by many approaches to whole program optimization and understanding [24]. Given a program call graph representing the possible callees at each call site, interprocedural analysis can typically provide valuable information for compiler optimization, program comprehension, and software maintenance tasks. Using a call graph, one can remove unreachable methods from the main method, replace dynamically dispatched method calls with direct method calls, inline method calls for which there is a unique target, and perform more sophisticated optimizations. In the context of software maintenance work, interprocedural analysis involving program call graph representation is also typically used in software testing [26, 27], bug finding [22, 25], change impact analysis [18, 29] and other activities.

During system evolution, software change is an essential operation that either introduces new functionalities or fixes bugs in the existing system, or modifies the former implementation if the requirements were not correctly addressed. As changes occur during the software life cycle, the call graph representation for the whole program may also change. Particularly, in an object-oriented program, due to the extensive use of sub-typing and dynamic dispatch, the nontrivial combination of small changes may affect the whole call graph. Though many call graph analysis approaches [8, 10, 13, 19, 24] have been presented in the literature, a common characteristic of these techniques is that they need to perform a *global exhaustive analysis*, i.e., they analyze the whole program code to construct the call graph. Given the unnecessary cost of computing a call graph with sufficient precision every time when change occurs, developing an incremental call graph construction algorithm with acceptable cost is desirable.

Aspect-Oriented Programming (AOP) [16] has been proposed as a technique for improving separation of crosscutting concerns in software design and implementation. A typical 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. When aspect-oriented features are added to an object-oriented program, or when an existing aspect-oriented program is modified, the existing call graph representation needs to be updated correspondingly. However, the existence of aspectual features complicate the analysis, since it can change dramatically the behavior of the original code as well as its call graph structure - e.g. without any change to the original Java code, an aspect can arbitrarily change the pre- and post- conditions of many methods and thus change the calling relationship between the call graph nodes. Moreover, the inherent intricacies of AspectJ semantics, such as *intertype declaration*, make it even more complex to analyze the calling relationship and perform incremental reanalysis for AspectJ software.

In this paper, we present a new source-code-level (static) call graph construction approach for AspectJ software. Unlike the previous approaches that rely on global analysis of the whole program, we *reuse* results from previous analysis to update the call graph in an attempt to perform an amount of work proportional to the source changes. We assume that a call graph has been exhaustively constructed for the initial software version, and after a session of source modifications, the incremental call graph construction algorithm is invoked according to program changes without global reanalysis. In our approach, given the updated and initial AspectJ software versions, we first decompose the source changes between these two versions into a set of *atomic change* representations [25, 29], which captures the semantic differences. Then, we exploit the relationship between *atomic changes* and *AspectJ call graph*, to update the initially constructed graph. As a result, a large portion of unnecessary reanalysis can be avoided and significant savings are observed in our experimental evaluation.

To our best knowledge, our work is the first attempt to address the incremental call graph reanalysis problem for AspectJ software. Our main contributions are threefold: ***(1)*** we choose a widely used call graph construction algorithm CHA [10] as basis, and present its corresponding call graph incremental reanalysis algorithm, ***(2)*** a call graph incremental renalysis tool that implements this algorithm using the *ajc* AspectJ compiler [2], and ***(3)*** an experimental study on 24 versions of eight AspectJ benchmarks. The results indicate our incremental algorithm can effectively reduce the cost of call graph construction.

```
class A {                           public class Main{
  public int i, j;                    public static void main(){
  public void m() {i++;}                A a = new A();
  public void n() {j++;}                B b = new B();
}                                       fun1(b);
                                        fun2();
class B extends A { }                 }

class C extends B {                   static void fun1(A a){
  public void m(){ i = i+3; }           a.m();
  public void n(){                      a.n();
    if(i > 0)  j = j+2; m();          }
  }
}                                      static void fun2(){
                                        B b2 = new C();
                                        b2.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++;}
}
```

Figure 1. A sample AspectJ program.

2. Background

We next use an example to briefly introduce the background of AspectJ semantics and the atomic change representation, which is the foundation of our reanalysis algorithm.

2.1. AspectJ Semantics

Figure 1 shows a small AspectJ program containing classes `A`, B, C, and aspect `M`. Here, we assume there is a sequence of edits to the original program in Figure 1. The edits are all new added and marked by underline.

A *join point* in AspectJ is a well-defined point in the execution that can be monitored - e.g. a call to a method, method body execution, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these compositions. *Advice* is a method-like mechanism that consists of instructions that execute *before*, *after*, or *around* a pointcut. An *aspect* is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. Moreover, an aspect can use an *intertype* construct to introduce methods, attributes, and interface implementation declarations into classes.

Example: in Figure 1, pointcut `callPoints` contains a join point when `C.n()` is executed if the runtime this object type is `A`. The aspect `M` in Figure 1 declares an advice, which is used to increase the value of `a.j` by one after each time `C.n()` is executed. Aspect `M` also defines an intertype method `B.m()`, which overrides the existing `A.m()`.

The AspectJ implementation ensures that the aspect and base code run together in a properly coordinated fashion. A key component is an *aspect weaver*, which ensures that applicable advice runs at appropriate join points. More information about AspectJ can be found in [3].

2.2. Atomic Change Representation

In our previous work [29], we identified a catalog of atomic changes for AspectJ programs shown in Table 1. Those atomic changes represent the source modifications at a coarse-grained model, which is amenable to analysis. Most of the atomic changes in Table 1 are self-explanatory except for **AIC**. Atomic change **AIC** captures the *advice invocations* changes. It reflects the semantic differences between the original and the edited programs; and indicates that the advice invoking at the certain join points has been

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

changed. The **AIC** changes are generated in situations where <advice, join point> pairs are added or removed as a result of source code changes. The formal definition of AIC is shown as follows:

$$\begin{aligned}\textbf{AIC} = \{<j,a>|<j,a> \in \\ ((J \times A - J' \times A') \cup (J' \times A' - J \times A))\}\end{aligned} \quad (1)$$

where J and A are the sets of join point and advices in the original program, and J′ and A′ are the sets of join point and advices in the modified program. J × A denotes the matched join points and advice tuple set in the original program, while J′ × A′ denotes the matched tuple set in the updated program version. Apart from aspect code changes, base code changes in Java are defined in [18] and [20].

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. Those semantic dependence rules are summarized in [18, 29].

Example: Figure 2 shows the atomic changes with their dependence relationships inferred from 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.

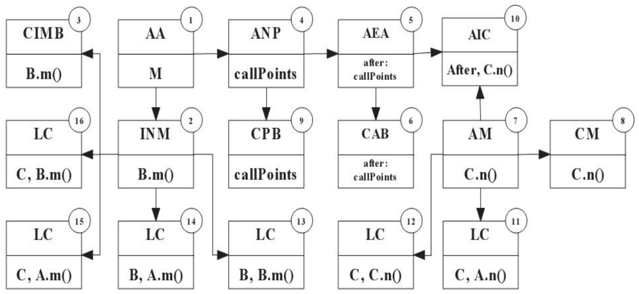

Figure 2. Atomic changes inferred from the sample program, with their dependence relationships

3. Call Graph Foundations

We next present the call graph representation used in this paper for Java and AspectJ programs briefly.

3.1. Call Graph for Java

In this paper, we focus on the static call graph analysis technique, that is, the objective of our analysis is to determine a call graph at compile time. One of the most important features of object-oriented programming languages like Java is the dynamic dispatch of methods based on the run-time type of an object. As for Java and other object-oriented languages, an important optimization problem is to statically determine what methods can be invoked by virtual method calls. We choose one efficient and widely used call graph construction algorithm, named *class hierarchy analysis* [10] (and its variant [15] for AspectJ) as our analysis foundation.

Class Hierarchy Analysis (CHA) is a standard method for conservatively estimating the run-time types of calling receivers [10]. In CHA, if a method is reachable, and a virtual call `a.f()` occurs in the body of this method, then every method with name `f` that is inherited by a subtype of the static type of `a` is also reachable.

Example: for the source in Figure 1, when using CHA algorithm to construct the call graph, method `fun1(A a)` will have three callers `A.m()`, `B.m()`, and `C.m()`. Similarly, `fun2()` will have two callers `B.m()` and `C.m()`, because `b2` has a static declared type `B`.

3.2. Call Graph for AspectJ Software

We build aspect-aware call graph [15] as follows. We employ CHA algorithm to construct call graph of the base code. For the aspect code, we consider the advice as a method-like node, namely advice node, with matching relationship represented by an edge from the join point (we

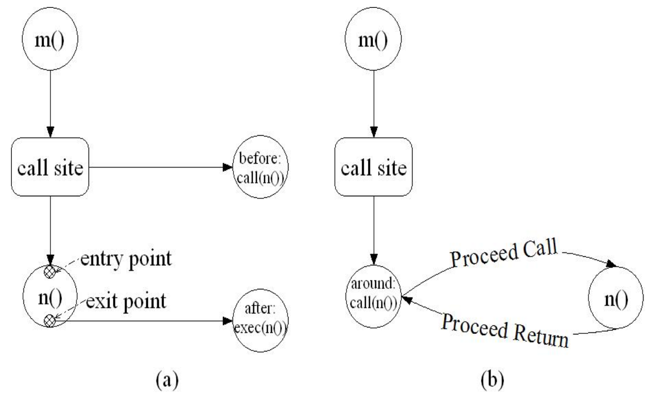

Figure 3. (a) Call graph for before and after advice, and (b) Call graph for around advice.

call such kind of edge a *shadow edge*). The complete call graph of AspectJ software is formed after the call graph of aspect code is *connected* into the base code call graph using the join point matching information.

Example: we take three typical aspectual constructs *before*, *after*, and *around* to show the basic idea of aspect-aware call graph construction.

***before* and *after* advice.** We add calling edges *before* or *after* the corresponding call site[1] to model such implicit calls introduced by AOP. For the following code snippet:

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

Its corresponding call graph is shown as Figure 3(a). Notice that we treat `call` and `execution` join point differently. The former is invoked at the call site while the latter is invoked at the entry or exit point of the advised method, that is, the calling edge of `execution` join point is connected to the method it weaves but not the call site.

***around* advice.** The `around` advice is more complex than `before` and `after` advices. We use two additional call sites to represent `proceed` call and `proceed return` nodes. For the following code snippet:

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

Its corresponding call graph shows in Figure 3(b). We use two kinds of calling edges (marked by *Proceed Call* and *Proceed Return*) to represent the `proceed` construct in AspectJ.

[1]Call site is a dummy node in our call graph representation which means there is a call relationship between caller and callee, and is used to differentiate `call` join point and `exec` join point.

4. Incremental Reanalysis Algorithm

In this section, we present our incremental reanalysis approach. In our approach, we first classify atomic changes based on their different effects on call graph structure in Section 4.1. We then match each affecting atomic change to corresponding call graph node or edge, and update the initial constructed graph in Section 4.2. A pseudo code of our algorithm is given in Section 4.3. Finally, we give a step by step example in Section 4.4.

4.1. Atomic Change Classification

To facilitate our analysis, we first classify atomic changes into the following five categories.

Method-related changes. There are three kinds of method-related atomic changes, named **AM**, **DM**, and **CM**. Intuitively, adding or deleting a method will affect the call graph structure. For the **CM** change, we further classify it into three subcategories *CM-Add*, *CM-Del*, and *CM-Local*. *CM-Add* is the **CM** change which adds new method call statements in its body; *CM-Del* is the **CM** change which deletes an existing method call statement in its body; and *CM-Local* is the **CM** change which only has local source change. We will ignore all *CM-Local* changes during analysis. Note that a **CM** change could be both a *CM-Add* and *CM-Del* category, since it can add and delete method call statements simultaneously during evolution. In aspect code, there are also three kinds of method-related changes, named **INM**, **DIM**, and **CIMB**. Similarly, we classify **CIMB** change into three subcategories *CIMB-Add*, *CIMB-Del* and *CIMB-Local*; in which *CIMB-Local* will not affect the call graph structure.

Advice-related changes. There are three kinds of advice-related changes, named **AEA**, **DEA**, and **CAB**. Like the **CM** change, we classify **CAB** into three subcategories named *CAB-Add*, *CAB-Del*, and *CAB-Local*; in which *CAB-Local* does not affect the call graph.

Dynamic dispatch changes. We preserve the precision of CHA algorithm during the incremental reanalysis by exploring the **LC** changes. According to the definition [20], **LC** abstracts any kind of source edits that would affect dynamic dispatch behavior, including adding or deleting methods, and adding or deleting inheritance relations. To facilitate our analysis, we borrow the triple *LookUp* [20] in the form of <*runtimeReceiverType*, *staticMethodSignature*, *actualMethodBound*>. For example, in a triple like <`C`, `A.f()`, `B.f()`>, `A` is an ancestor of `B` or `A` is `B`, `A.f()` is a method declared in the hierarchy, `B` is the nearest superclass of `C` containing a definition of method `f()`, `C` is the run-time type of the receiver, `A.f()` is the method that is statically referred to in the method call, and `B.f()` is the actual bounded method. In our example in Figure 1, we have a *LookUp* <C, A.n(), A.n()> in the old version. After adding a method n() to class C, the new *LookUp* triple become <C, A.n(), C.n()>.

Dynamic dispatch behavior changes might lead to adding or deleting edges in the virtual call site. We classify **LC** in to two categories: *LC-Add* and *LC-Del*. *LC-Add* represents the **LC** changes which will add edges into the call graph, while *LC-Del* represents the **LC** changes which will delete edges from the call graph. The formal definitions of LC-Add and LC-Del show as follows.

$$\begin{aligned}
&\mathit{LookUp}_{preserve} = \{<C_{old}, A.f(), B.f()>| \\
&\quad <C_{old}, A.f(), B.f()> \in \mathit{LookUp}_{old} \wedge \\
&\quad <C_{new}, A.f(), B.f()> \in \mathit{LookUp}_{new}, \\
&\quad \forall C_{old} \in \text{classes in } v_{old}, C_{new} \in \text{classes in } v_{new}\} \\
&\mathit{LC\text{-}Add} = \mathit{LookUp}_{new} - \mathit{LookUp}_{old} \\
&\mathit{LC\text{-}Del} = \mathit{LookUp}_{old} - \mathit{LookUp}_{new} - \mathit{LookUp}_{preserve}
\end{aligned}$$

In the equation of $\mathit{LookUp}_{preserve}$, C_{old} is the class set in the old version, and C_{new} is the class set in the new version. $\mathit{LookUp}_{preserve}$ contains those unchanged *LookUp* triples during evolution, while *LC-Add* and *LC-Del* represents those new added and deleted ones, respectively.

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 [29]. As shown in section 2.2, all the information related to these changes are reflected by **AIC** change. We classify **AIC** into two categories, *AIC-Add* and *AIC-Del*, which represents the newly added invocation and deleted <jonpoint, advice> pairs during software evolution, respectively.

Using the equation 1 and terms in Section 2.2, we define *AIC-Add* and *AIC-Del* as follows:

$$\begin{aligned}
&\mathit{AIC\text{-}Add} = \{<j,a>|<j,a> \in (J \times A - J' \times A')\} \\
&\mathit{AIC\text{-}Del} = \{<j,a>|<j,a> \in (J' \times A' - J \times A)\}
\end{aligned}$$

Ignored changes. We ignore other changes (not mentioned above) that will not directly affect the call graph structure, like **DI**, **CI**, **AA**, and **AC**. Note that changes to the pointcut body are also not considered, because their effects to the call graph could be reflected by **AIC** changes. Similarly, class hierarchy-related changes like **AHD** and **DHD** will not be handled specially, since their effects are reflected by **LC** change.

4.2. Match Atomic Change to Call Graph

We next match each change to the initial constructed call graph, to decide which parts need to be updated. The matching process consists of the following four parts.

AM, INM, and AEA changes. Changes like **AM**, **INM**, and **AEA** require us to add new nodes to the original call graph, while changes like **DM**, **DIM**, and **DEA** indicate that we need to delete existing nodes and their incoming/outgoing edges. For example, the new added intertype method B.m() of aspect M in Figure 1 requires us to add a corresponding node to the initial graph.

CM, CIMB, and CAB changes. Changes like **CM**, **CIMB**, and **CAB** require us to update corresponding method, intertype method, or advice nodes. Here, we use the category classification information (e.g. *CM-Add*) to find what kind of changes (e.g. adding or deleting call relations) have taken place. Then, we add or delete corresponding edges in call graph. For example, in Figure 1, method C.n() has been changed. A method call to C.m() has been added in its method body. We classify this change to be a *CM-Add* category, and then add a new edge from C.n() to C.m() in the call graph.

LC changes. We use the *LC-Add* and *LC-Del* change sets to update corresponding virtual call sites. For each tuple <C,A.f(),B.f()> ∈ *LC-Add*, our algorithm finds all graph nodes which call A.f() and add a new callee node B.f() for them (if this node has not been added yet). For each tuple <C,A.f(),B.f()> ∈ *LC-Del*, our algorithm finds all callers which call B.f() and deletes their callee node A.f(). For example, in Figure 1, we have one **LC**(<C,A.n(),C.n()>) change, which is classified as *LC-Add* category. Our algorithm finds the graph node Main.fun1() is the caller of A.n(). Then, we add a new callee node C.n() for node Main.fun1().

AIC changes. The **AIC** change, representing as tuples <*join point*, *advice*>, indicates that the *advice* invoking at the above *join point* has been changed. In such case, we find each affected join point, then add or delete shadow edges between join point and advice. For example, in Figure 1, suppose that we also changed **pointcut** callPoints(A a): **execution**(* C.n()) && **this**(a) to **pointcut** callPoints(A a): **execution**(* A.n()) && **this**(a). We will get two **AIC** changes, in which **AIC**<A.n(), after advice> belongs to *AIC-Add* category and **AIC**<C.n(), after advice> belongs to *AIC-Del*. Our algorithm finds the graph nodes A.n() and C.n() which correspond to the affected join points. Then, we add a shadow edge from A.n() to the after advice and delete the edge between C.n() and the after advice.

4.3. Incremental Reanalysis Algorithm

This section presents our incremental reanalysis algorithm. Figure 4 shows the pseudo code of the algorithm.

The algorithm takes two inputs: an original call graph and a set of atomic changes. The algorithm first uses a helper function *classify* to classify atomic change set and prune out all the ignorable changes.

We first add new nodes to the initial constructed graph, representing **AM**, **INM**, and **AEA** changes, and then delete existing graph nodes and edges according to the **DM**, **DIM**, and **DEA** changes. This part corresponds to lines 4 to 9 in Figure 4. Here, *addNode* and *deleteNode* are helper functions which simply add and delete nodes in the original graph.

After that, we update the incoming or outgoing edges for each modified node (represented by **CM**, **CIMB**, and **CAB** changes). For each node, we add or delete its callers or callees according to the change category (lines 10 to 18 in Figure 4). Here, we use helper functions *addCall* and *deleteCall* to perform such actions.

Afterwards, we update each virtual call site using **LC** (*LC-Add* and *LC-Del*) changes. We use **LC** to preserve the precision of CHA algorithm. In lines 21 and 23 of Figure 4, we use helper functions *addDynamicDispatch* and *deleteDynamicDispatch* to deal with dynamic dispatches. Each function takes three parameters: `A` is the static referred type, `B` is the dynamic bounded type, and `m` is the virtual method node to be updated.

Finally, we update the shadow edges in the initial call graph using the **AIC** change. An **AIC** change contains all weaving information, including the adding and deleting relationship between advice and join point. In lines 26 to 34, we use helper functions *addAdvice* and *deleteAdvice* to update shadow edges between join points and advice nodes. The first input `methodSet` of the function contains all methods which are matched by a join point. The second input `advice` is the corresponding advice which is invoked at that join point.

4.4. Put It Together: A Full Example

With respect to our example in Figure 1, we demonstrate the whole reanalysis process in Figure 5. Figure 5(a) is the initial constructed call graph before source editing, and Figure 5(e) is the updated call graph after reanalysis. For simplicity, we omit the call site nodes and do not distinguish method node and advice node, as well as shadow edge and normal call graph edge. The red shadow nodes in Figure 5 are newly added, while the red dashed edges are the updated ones. In the first step, three new nodes which represent `C.n()`, `after(A a):callPoints`, and `B.m()` are added (Figure 5(b)). In the second step, an new edge from `C.n()` to `C.m()` is added by a *CM-Add* change (Figure 5(c)). In the third step, three *LC-Add* changes lead to three more virtual call edges, namely `fun1(A a)` to `B.m()` and `C.n()`, `fun2()` to `B.m()` (Figure 5(d)). Finally, a new shadow edge is added from method node `C.n()` to advice `after(A a):callPoints`.

```
Algorithm BuildIncrementalCG

    Input: the orginal call graph: cg, a set of
           atomic changes: acSet.
    Output: the updated call graph: cg'.

 1: function INCREMENTALCG($cg, acSet$)
 2:     $cg' \leftarrow cg$
 3:     classify($acSet$);
 4:     for all $ac \in AM \cup INM \cup AEA$ do
 5:         addNode($cg', ac$)
 6:     end for
 7:     for all $ac \in DM \cup DIM \cup DEA$ do
 8:         deleteNode($cg', ac$)
 9:     end for
10:     for all $ac \in CM \cup CIMB \cup CAB$ do
11:         $caller \leftarrow$ getCaller($ac$)
12:         $callee \leftarrow$ getCallee($ac$)
13:         if $ac \in$ CM-Add $\cup$ CIMB-Add $\cup$ CAB-Add then
14:             addCall($cg', caller, callee$)
15:         else
16:             deleteCall($cg', caller, callee$)
17:         end if
18:     end for
19:     for all $ac \in$ LC<C,A.m(),B.m()> do
20:         if $ac$ is LC-Add then
21:             addDynamicDispatch($A, B, m$)
22:         else
23:             deleteDynamicDispatch($A, B, m$)
24:         end if
25:     end for
26:     for all $ac \in AIC$ do
27:         $advice \leftarrow$ getAdvice($ac$)
28:         $methodSet \leftarrow$ getMatchedM($ac$)
29:         if $ac$ is AIC-Add then
30:             addAdvice($methodSet, advice$)
31:         else
32:             deleteAdvice($methodSet, advice$)
33:         end if
34:     end for
35:     return $cg'$
36: end function
```

Figure 4. Call Graph Incremental Reanalysis Algorithm

5. Empirical Evaluation

To investigate the effectiveness of our proposed approach, we implemented the analysis algorithm on top of the *ajc* compiler [2]. We performed an experimental study on AspectJ benchmarks, ranging from hundreds to thousands of lines of code. The empirical study indicates that we are able to achieve an average 76% decrease in the call

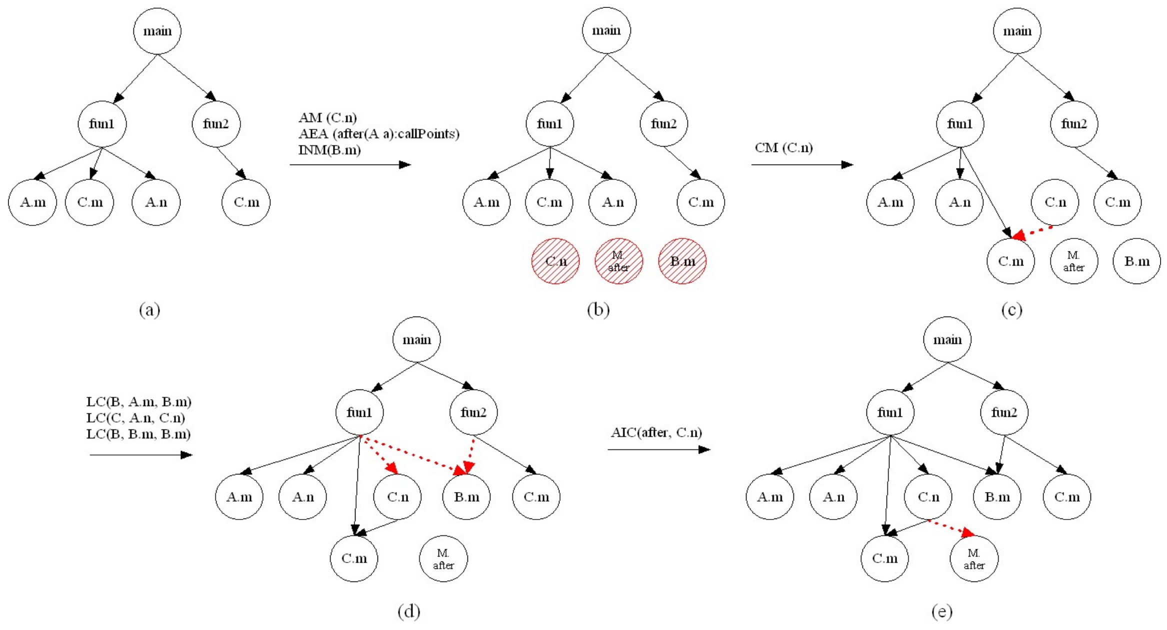

Figure 5. Incremental call graph reanalysis: a full example.

graph reanalysis cost, compared with exhaustive analysis.

5.1. Subject Programs

We use 24 versions of eight AspectJ benchmarks collected from a variety of sources as our subject programs. The first three and the spacewar example are included in the AspectJ compiler example package [2]. Other programs are obtained from the abc benchmark package [1]. This group of benchmarks have also been widely used by other researchers to evaluate their work [12,26,27]. For each program, we make the first version v_1 a pure Java program by removing all aspectual constructs. Table 2 shows the number of lines of code in the original program (#Loc), the number of versions (#Ver), the number of methods (#Me), and the number of shadows (#Shad).

Programs	#Loc	#Ver	#Me	#Shad
Quicksort	111	3	18	15
Figure	147	4	23	5
Bean	199	3	12	8
Tracing	1059	4	44	32
NullCheck	2991	5	196	146
Lod	3075	2	220	1103
Dcm	3423	2	249	359
Spacewar	3053	2	288	369

Table 2. Subject Programs

5.2. Procedure

To evaluate our approach, we take each successive version pair of an AspectJ benchmark (i.e., v_1 and v_2, v_2 and v_3, etc) and the call graph of the initial version (i.e., the v_1 version of v_1, v_2 pair) as the input of our algorithm. For each input, we first decompose the changes between two program versions into a set of atomic changes, and then use the atomic changes to update the initial constructed graph. The updated call graph is the output as our analysis result.

5.3. Results

In our previous work on change impact analysis [29] using the same subject programs, we found that on average 44% of the call graph nodes have been affected. Here, we count the number of the atomic changes and updated nodes/edges between two successive versions, and compare the effectiveness between exhaustive CHA algorithm [10] and our incremental approach in terms of construction time. Experimental results are shown in Table 3 and 4. In these two tables, each AspectJ program version is labelled with its number - e.g. 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 row v_n.

5.3.1. Updated Nodes and Edges. In Table 3, columns *Nodes* and *Edges* indicate the number of call graph nodes and edges, respectively. Column *Atomic Changes* and *Related Changes* indicate the number of all the atomic changes between two successive versions, and those atomic changes

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

Table 3. Updated call graph nodes and edges of the incremental approach

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

Table 4. Analysis time of exhaustive analysis and incremental approach (excluding compilation time)

that affect the call graph, respectively. The number in columns *Updated nodes* and *Updated edges* are defined as follows to indicate the effectiveness.

$$N_{updated} = \frac{N_{add} + N_{change} + N_{delete}}{N_{new} + N_{delete}} \times 100\% \qquad (2)$$

$$E_{updated} = \frac{E_{add} + E_{delete}}{E_{new} + E_{delete}} \times 100\% \qquad (3)$$

In this two equations, N_{add}, N_{change},N_{delete} represent the number of graph nodes which are added, changed, or deleted. If a node's incoming or outgoing edges are changed, we say it is changed. E_{add} and E_{delete} represent the number of added and deleted graph edges. N_{new} and E_{new} are the number of nodes and edges in the call graph of the updated version (i.e., the v_n version of v_{n-1}, v_n pair).

We can observe that our incremental approach updates 51% of the call graph nodes and 49% of the call graph edges on average.

5.3.2. Construction Time. Table 4 shows the construction time of both exhaustive analysis and incremental approach. Column *Exhaustive Time* is the total time cost when using original CHA algorithm, while column *Incremental Time* indicates the time cost of our incremental approach. The incremental algorithm cost is shown as a percentage compared with the exhaustive algorithm.

We can observe that the average call graph construction time of our incremental approach is 24% of the exhaustive analysis cost. But for L2, we reduce only a small portion of the time cost.

5.4. Discussion

In our experiment, the call graphs constructed by our incremental algorithm are the same with graphs constructed by the exhaustive algorithm in most cases. The only exception cases are the L2, N2, and D2 groups. For example, in L2, there are 274 call sites and 326 shadow edges in the call graph constructed by the exhaustive algorithm, while there are 261 call sites and 315 shadow edges in the call graph constructed by the incremental algorithm. The discrepancy is caused by the default constructor changes, especially changes in the compiler-added `<init>()` constructor. Such compiler-added constructors are not counted in the atomic change set, but appear as nodes in the call graph representation. Another cause is due to the pointcut that crosscuts such default constructors, like the pointcut declaration **`call`**(*.**`new`**(..)) in L2. In our implementation, we ignore such compiler-added *shadow edges*.

From Table 3, we can find more related atomic changes will lead to more node and edge updates in most cases. However, there are some exceptions: in F3, 62 related changes result in 81% node update, while in F4, 43 changes lead to 89% update of node; in N4 and N5, one atomic change results in different amount of updates. We think this is reasonable: some changes may only affect a small portion of the graph. From Table 4, we can see our incremental algorithm can result in a great improvement, except for L2. Because the effectiveness of our approach is not only determined by the number of the related changes, but also how many edges these changes have affected, if there are lots of related changes and most of them affect many edges, the improvement will be low. The evidence to support our assumption can be found in Table 3, only L2 has both high updated edges (77%) number and a large amount of related changes (1492).

5.5. Threats to Validity

Like any empirical evaluation, this study also has limitations which must be considered. Although we have experimented 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 cannot claim that these experiment results can be necessarily generalized to other programs. On the other hand, threats to internal validity maybe mostly lie in the call graph differences between exhaustive and incremental algorithm. As discussed in Section 5.4, the call graph built by our algorithm differ slightly from the exhaustively constructed one. Investigating the impact of such difference on client analyses would be our future topics.

6. Related Work

We next discuss some related work in areas of call graph construction and incremental analysis techniques.

Call graph construction for AspectJ software. Call graph construction has been an important area of research within the programming language and software engineering community. Various call graph analysis approaches [8, 10, 13, 19, 24] have been proposed in the literature. However, most of them are focused on procedural or object-oriented context. Rinard et al [30] proposed a control flow graph in the classification system for AspectJ programs. They used a lightweight call graph representation to model before, after, and around advices based on program transformation. Sereni and Moor [21] also present a simple call graph for a simple AOP language, which considers that basic concepts of invocation relationship of aspects. In [15], Huang proposed an algorithm for constructing aspect-aware call graphs. He also presented the corresponding graph construction tool support together with a set of evaluation results. Our work is based on Huang's call graph representation for AspectJ software. However, all the above approaches use an exhaustive program analysis to build the AspectJ call graph. In this paper, we reuse the initial constructed call graph along with change information to incrementally update the program call graph, to eliminate unnecessary analysis cost.

Incremental and demand driven analysis. There is also a rich body of work on incremental and demand driven program analysis. Iterative-based, interval-based, and hybrid incremental data flow analysis algorithms were first developed for intraprocedural data flow analysis [6, 23, 28].

Little work has been carried out in incremental analysis within the object-oriented and aspect-oriented context. Perhaps the most similar work to ours is Souter and Pollock's approach [23] to incrementally updating the call graph of Java programs. However, they use a relatively expensive approach - Cartesian Product Algorithm [4] as the analysis basis and transform each source editing as *adding* and *deleting* a call site. As in our work, we use *atomic change* representation not just *call site deletion and insertion* actions to capture the source changes. The *atomic change*s can be used to reflect the semantic differences of source edits and update the initial graph. On the other hand, we use a more efficient algorithm CHA [10] to construct base code call graph and consider the unique aspectual features.

Demand driven analysis techniques have also been studied by many researchers. In particular, these analysis techniques [5, 6, 11, 14] are for compiler optimization of object-oriented programs. Agrawal developed a demand driven call graph construction algorithm that solves the problem of computing the set of procedures potentially called at a particular call site without computing this information for every call site [6]. This algorithm updates information for a single call site, and thus is suggested for scenarios in which precise call graph information is only needed at certain call sites on a demand basis, such as program slicing. In contrast, besides we can handle the specific aspectual features, in our approach, a precise call graph has already been built, and it is desired to be updated after a session of changes occur. As pointed out in [7], demand driven call graph construction algorithm is only concerned with narrowing type sets, whereas incremental call graph construction in response to changes must handle the possibility of both narrowing and widening reaching type sets in order to avoid precision losses.

7. Concluding Remarks

In this paper, we presented an incremental call graph reanalysis algorithm for AspectJ software. We use *atomic change* representation to capture the semantic differences between two program versions. We also explore the relationship between *atomic change*s and AspectJ *call graph*, and update the initial constructed call graph incrementally. The main advantage of our approach is to reuse the existing analysis result, and therefore, eliminate a large portion of unnecessary cost.

We experimented the reanalysis algorithm based on 24 versions of AspectJ benchmarks. The result indicated, for the subject programs we investigated, our approach can significantly reduce as much as 75% analysis cost to an exhaustive analysis. As our future work, we would like to apply the basic idea of our reanalysis approach to other call graph [24], control flow graph [9, 15, 17], and system dependence graph [30] analysis techniques, to reduce the analysis cost in software maintenance.

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).

References

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

[2] AspectJ Development Tools (AJDT). `http://www.eclipse.org/ajdt//`.

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

[4] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26, London, UK, 1995. Springer-Verlag.

[5] G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *ICSM*, pages 453–462, 1999.

[6] G. Agrawal. Demand-driven construction of call graphs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 125–140, London, UK, 2000. Springer-Verlag.

[7] G. Agrawal, J. Li, and Q. Su. Evaluating a demand driven technique for call graph construction. In *Computational Complexity*, pages 29–45, 2002.

[8] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM.

[9] M. L. Bernardi and G. A. D. Lucca. An interprocedural aspect control flow graph to support the maintenance of aspect oriented systems. In *ICSM*, pages 435–444, 2007.

[10] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[11] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–48, New York, NY, USA, 1995. ACM.

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

[13] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, 1997.

[14] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 104–115, New York, NY, USA, 1995. ACM.

[15] S. Huang and J. Zhao. AJFlow: A framework and tool support for control flow analysis of AspectJ programs. In *AOAsia 2008: Proceedings of the 4th Asian Workshop on Aspect-Oriented Software Development*, 2008.

[16] 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.

[17] R. M. Parizi and A. A. A. Ghani. AJcFgraph - AspectJ control flow graph builder for aspect-oriented software. In *International Journal of Computer Science*, pages 170–181, 2008.

[18] 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.

[19] B. G. Ryder. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3):216–226, 1979.

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

[21] D. Sereni and O. de Moor. Static analysis of aspects. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39, New York, NY, USA, 2003. ACM.

[22] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao. XFindbugs: eXtended FindBugs for AspectJ. In *PASTE'08*, pages 70–56, 2008.

[23] A. L. Souter and L. L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 682, Washington, DC, USA, 2001. IEEE Computer Society.

[24] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM.

[25] O. C. Xiaoxia Ren and B. G.Ryder. Identifying failure casues in Java programs: An application of change impact analysis. *IEEE Trans. Softw. Eng.*, 32:718– 732, 2006.

[26] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Proc. 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 190–201, March 2006.

[27] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 65–74, Washington, DC, USA, 2007. IEEE Computer Society.

[28] J.-S. Yur, B. G. Ryder, W. A. Landi, and P. Stocks. Incremental analysis of side effects for C software system. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 422–432, New York, NY, USA, 1997. ACM.

[29] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *IEEE ICSM 2008*, pages 87–96, 2008.

[30] J. Zhao and M. Rinard. Constructing system dependence graphs for aspect-oriented programs. Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.