The abc Group

Static Aspect Impact Analysis

abc Technical Report No. abc-2007-5

Dehua Zhang and Laurie Hendren
School of Computer Science
McGill University
Montréal, Québec, Canada

November 14, 2007

Contents

1 Introduction **3**
1.1 Contributions . . . 3

2 Classification **4**
2.1 Examples . . . 4
2.1.1 Bank . . . 4
2.1.2 Source Code Repository . . . 6
2.2 State Impacts . . . 7
2.3 Computation Impacts . . . 7
2.3.1 Invariant advice . . . 8
2.3.2 Variant advice . . . 8

3 Impact Analyses **9**
3.1 Analyses Workbench . . . 9
3.2 Considerations . . . 9
3.3 State Impact Analyses . . . 11
3.3.1 Direct State Impacts . . . 11
3.3.2 Indirect State Impacts . . . 12
3.4 Computation Impact Analyses . . . 12
3.4.1 Exact-proceed analysis . . . 12
3.4.2 Conclude Impact . . . 15

4 Examples **15**
4.1 Bank . . . 15
4.2 Source Code Repository . . . 16
4.3 Glass and Table . . . 16
4.4 Classification of aspects or advices . . . 17

5 Related Work **17**

6 Conclusions and Future Work **18**

List of Figures

1 Impact analyses in abc, extended from [dM04] 10
2 Pseudo-code for state impact analysis 11
3 Pseudo-code for computation impact analysis 13

List of Tables

I Classification of computation impacts 8

Abstract

One of the major challenges in aspect-oriented programming is that aspects may have unintended impacts on a base program. Thus, it is important to develop techniques and tools that can both summarize the impacts and provide information about the causes of the impacts. This paper presents static impact analyses for AspectJ.

Our approach focuses on two kinds of impacts, *state impacts* which cause changes of state in the base program, and *computation impacts* which cause changes in functionality by adding, removing or replacing computations of the base program.

We provide a classification scheme for these two kinds of impacts and then develop a set of static analyses to estimate these impacts. A key feature of our approach is the use of points-to analysis to provide more accurate estimates. Further, our analysis results allow us to trace back to find the causes of the impacts.

We have implemented our techniques in the AspectBench compiler and provide examples of impact results.

1 Introduction

Aspect-oriented programming (AOP) introduces aspects as language constructs that address cross-cutting concerns [SW07]. Aspects can observe, alter or augment the behavior of base programs. Although this functionality is very powerful, it is also possible that aspects break encapsulation or impact on the based program in unintended ways. The purpose of this paper is to provide static analyses that can summarize impacts and help programmers locate the causes of impacts in AspectJ programs.

AspectJ is a popular AOP language which is defined as a convenient extension of Java. An aspect defined through AspectJ can modify the base program state by writing to fields, or can change the program's execution by adding to, substituting, repeating or eliminating the computation in base program. Complex interactions between the base program and aspects can make AspectJ programs difficult to understand and maintain. The possibility of obliviously and globally changing the behavior of the base program [Stö03] may lead to undesired and unexpected impacts. In particular, AspectJ users may find that aspects interact with classes, components or data structures in a way that was not anticipated. Thus, even though many programmers have good uses for aspects, the uncertainty about the impacts of the aspects on the base code can limit adoption of AspectJ. Therefore, techniques and tools that can both analyze impacts of aspects on base programs and summarize the causes of impacts are desired.

Since the behavior of a program depends on both the program state and which computations are executed, we classify impacts into two categories, *state impacts* which indicate changes of state in the base program, and *computation impacts* which indicate changes to which computations are performed. We further classify computation impacts into *invariant* or *variant*. We expect most advice to actually do something, so most advice will have a variant computation impact. Variant computation impacts come in four flavours, *addition*, *elimination*, *must-substitution* and *may-substitution* based on whether computations were added, eliminated or possibly/definitely substituted for different computations. Following this classification, we implemented a series of static analyses in the back-end of the AspectBench compiler, abc, to analyze impacts of aspects on both the state and computation of the base program so that the hidden impact of aspects are revealed. We analyze how fields of base classes are accessed to expose state impacts, and we analyze the effect of advice on computation by categorizing each advice into the appropriate computation impact category. Key features of our approach are that we make use of the points-to analysis available in abc to give a more precise analysis, and we use of analysis results to provide a more descriptive report of what causes the impacts.

1.1 Contributions

This paper studies the complicated interactions between advice and base programs in AspectJ. We make the following specific contributions:

- A concise classification of impacts based on state and computation changes caused by advice.

- Implementation of static analyses in the AspectBench Compiler to analyze and classify impacts based on our classification.
- An informative analyses report, which provides both impact information and the causes of impacts which can guide the programmer in understanding these impacts.
- Experience on three example AspectJ programs to illustrate how our classification and analyses can help programmer to understand program and fix bugs caused by improper designed advice.

The remaining sections are structured as follows. We first introduce our classification system and provide some motivating examples in Section 2. In Section 3, the algorithm and challenges of our analysis are presented and discussed. Section 4 provides some examples to demonstrate our technique and analyses. Finally, we give related work in Section 5, and conclusions and future work in Section 6.

2 Classification

In this section we outline our classification scheme. In Section 2.1 we first provide two small example applications which we refer to throughout the paper. We then define *state impacts* which capture when advice writes to fields of the base program in Section 2.2 and *computation impacts* which capture when advice causes computation to be eliminated, replaced or added in Section 2.3.

2.1 Examples

Before starting our classification discussion, we first present two example AspectJ programs: bank and source code repository. We will refer these two examples when explaining our classification, presenting our analyses and talking about our experience.

2.1.1 Bank

```
4  public abstract class Account {
5
6      protected Date lastVisit;
7      protected double money;
8      public final double FEE = 2;
9
10     public Account(double money) {this.money = money;}
11
12     public void debit(double m) {money = money - m;}
13
14     public void credit(double m) {money = money + m;}
15
16     public void fee() {money = money - FEE;}
17
18     public void transfer(Account other, double m) {
19         other.credit(m);
20         this.debit(m);
21     }
22 }
```

Listing 1: Account.java

```
3  public class CheckingAccount extends Account {
4
5      public CheckingAccount(double money) {
```

```
6           super(money);
7       }
8   }
```

Listing 2: CheckingAccount.java

```
3   public class Bank {
4
5       public static void main(String [] args)
6       {
7           Account acct1 = new CheckingAccount(200);
8           Account acct2 = new CheckingAccount(2000);
9
10          acct1.credit(300);
11          acct1.debit(200);
12          acct2.transfer(acct1, 200);
13      }
14  }
```

Listing 3: Bank.java

```
4   public aspect AccountAspect {
5
6       before(Account account) :
7           (execution (public void Account+.debit(double))
8               || execution(public void Account+.credit(double)))
9               && target(account) {
10          account.lastVisit = new Date();
11      }
12
13      after (Account account) :
14          execution (public void Account+.debit(double))
15              && target(account) {
16          account.fee();
17      }
18
19      void around() :
20          execution (public void Account+.transfer(Account, double)) {
21          System.out.println("Transfer starts at "+new Date());
22          proceed();
23          System.out.println("Transfer completes at "+new Date());
24      }
25  }
```

Listing 4: AccountAspect.aj

In the bank example, we are simulating a system developed in an aspect-oriented way. There is an abstract Account class (Listing 1) defining basic functions on account like debiting, crediting, transferring, and fee charging. The CheckingAccount class (Listing 2) simply extends the abstract Account class. The Bank class (Listing 3) only contains a main method, in which two CheckingAccount objects are initialized and three different transactions are performed. These three classes form our base program. In the aspect called AccountAspect (Listing 4), three advice declarations are given. The **before** advice records the current time as the last visit time. The **after** advice changes the fee after each debit or transfer transaction. The **around** advice measures the time taken to make a transfer.

2.1.2 Source Code Repository

```
3  public class SourceCodeRepository {
4
5      private String sourcecode;
6      private String user;
7      private String pass;
8
9      public SourceCodeRepository(String src) {
10         this.sourcecode = src;
11     }
12
13     private boolean login() {
14         if (user.equals("mcgill") && pass.equals("sable")) return true;
15         else return false;
16     }
17
18     public String getSrc() {
19         if (login())
20             return sourcecode;
21         else return null;
22     }
23
24     public void putSrc(String src) {
25         if (login()) {
26             this.sourcecode = src;
27         }
28     }
29
30     public static void main(String [] args) {
31         SourceCodeRepository repo = new SourceCodeRepository("foo");
32         System.out.println(repo.getSrc());
33         repo.putSrc("junk foo");
34         System.out.println(repo.getSrc());
35     }
36 }
```

Listing 5: SourceCodeRepository.java

```
3  public aspect SourceCodeRepositoryAspect {
4
5      boolean around() :
6          execution (boolean SourceCodeRepository.login()) {
7          return true;
8      }
9
10     void around(String src) :
11         execution (void SourceCodeRepository.putSrc(String))
12             && args(src) {
13         if (src.length() < 5) {
14             proceed(src);
15         } else {
16             System.out.println("Out of limit");
17         }
18     }
19 }
```

Listing 6: SourceCodeRepositoryAspect.aj

The Source Code Repository example is intended to demonstrate a legacy object-oriented system patched by aspects. In the original base program, in class SourceCodeRepository (Listing 5), it requires a check of username and password every time before a retrieve or store operation. However, let us assume that the repository becomes open-source, and no authentication is needed for accessing code, but a filter is needed to filter out junk programs. Therefore, in aspect SourceCodeRepositoryAspect (Listing 6), two advice declarations are defined. The first **around** advice captures the login method and always returns true to bypass the login step, and the second **around** advice captures the putSrc method and rejects the source code if the filter (in this case just implemented as a length check) returns false.

2.2 State Impacts

The base program of an AspectJ program is a Java program, which is object-oriented program. In the scope of an OO program, the program state is mainly defined by values of fields in classes. Therefore, our state impacts focus on how aspects can modify the fields of the base program. Because aspects interact with program at the granularity of advice, we define our state impacts in the granularity of advice too. Since an advice can change fields of base classes both directly and indirectly, we classify state impacts further into:

Direct state impacts: are impacts caused by advice modifying fields of base classes directly in the form such as *base.field = NewValue*. In the bank example, "account.lastVisit = new Date();"(Listing 4, line 10) in the **before** advice causes a direct state impact because it writes the field lastVisit in the class Account or its subclasses.

Indirect state impacts: are impacts caused by advice invoking methods which modify fields of base classes. In bank example, "account.fee();"(Listing 4, line 16) in the **after** advice causes indirect state impacts. If we check the method body of Account.fee(), we can see that the money field is modified, but the money field may belong to Account or its subclasses. Later in the paper, in Section 4.1, we will see that points-to analysis can give the precise estimation of the actual class that money belongs to.

Therefore, we define *state impacts* as program state change caused by advice modifying the value of fields of base classes directly or indirectly.

2.3 Computation Impacts

Applying advice to a the base program usually changes the computation performed. An advice will match a collection of shadows in the base program. **Before** and **after** advice can add computation before or after the shadow; however, **around** advice can have an impact on whether or not the shadow code executes, depending on how the body of the advice calls **proceed**. Thus, in order to handle **around** advice, we first introduce the concept of *exact-proceed*. We define an *exact-proceed* as a proceed call that fulfills the following three conditions:[1]

same arguments: the same argument values as found in the join point must be passed by the proceed call;

same return value: the value returned by *proceed* must be returned by the advice without modification; and

no abrupt exception: no exception stops the reachability of *proceed*.

The idea behind these conditions is that the *same arguments* and *same return value* conditions ensure that the original computation at the join point is executed and the same value returned, whereas the *no abrupt exception* condition ensures that the computation is always executed as it was in the base program.

We also use the concept of *live* and *dead* advice. For a specific program, we say that an advice is *live* if it matches at least one shadow and it is *dead* if it does not match any shadows.

Given these definitions we now define *invariant advice* and four flavours of *variant advice*.

[1]Similar conditions have been defined by Recebli [Rec05] and Rinard. et. al. [RSB04].

2.3.1 Invariant advice

We define an advice to be invariant if it adds no new computation to any shadow, nor removes any computation from any shadow. For **before** and **after** advice either: (a) the advice is *dead*, i.e. it doesn't match any shadows, or (b) the advice body is empty. For **around** advice, either: (a) the advice is dead, or (b) the body of the advice is composed of exactly one exact-proceed. Invariant advice is not very interesting and if we find invariant advice it is likely to indicate a bug in the program (for example, some AspectJ compilers give a warning when an advice doesn't match anywhere in the program).

	Variant				**Invariant**
	Addition	**Elimination**	**Must-Substitution**	**May-Substitution**	
before after	*live* and non-empty body				*dead* or empty body
around	*live* and at least one *exact-proceed* on every path, plus additional computation	*live* and empty body	*live* and and no *exact-proceed* on any path	*live* and at least one *exact-proceed* on one or more paths, but not on all paths	*dead* or exactly one *exact-proceed* with no additional computation

Table I: Classification of computation impacts

2.3.2 Variant advice

Variant advice is much more interesting and useful than invariant advice. The idea is that we want to know if the advice adds computation to matching shadows, eliminates code at shadows, or replaces code at shadows. We classify variant advice according into the following kinds of computation impacts:

Addition: After applying the advice (weaving), the matched shadows in the base program always execute unchanged, and new computation is added. Logging advice [Lad03] is a typical example. In the bank example, all three advice definitions have addition computation impacts.

Elimination: After applying the advice, matched computation in the base program is removed, and no new computation is added. Advice hiding method functionality often cause elimination impacts. In the source code repository example, the **boolean around** advice (Listing 6, line 5) has an elimination impact.

Must-Substitution: After applying advice, the matched computation in the base program does not execute at all, and new computation is added. In this case the advice replaces a functionality in the base program with a brand-new one. An example is an advice replacing an old algorithm with optimized algorithm.

May-Substitution: After applying advice, matched computation in the base program may or may not executed changed or unchanged depending on some conditions, and new computation is always added. In this case an advice may replace a functionality in the base program under certain conditions. An example is an advice that introduces a run-time check to determine if the old algorithm should be replaced by an different algorithm. In source code repository example, the **void around** advice (Listing 6, line 10) has may-substitution impact.

In Table I, we present a summary of our categorization expressed in terms of our definitions of live/dead advice and the definition of exact-proceed. Note that before and after advice is either invariant or has addition impact, because the original shadow code is never removed. However, around advice can have different impacts, depending on how *proceed* is used in the body of the advice. In the next section we discuss the analyses used to determine the classifications.

3 Impact Analyses

We first briefly review the underlying analyses framework: abc, the AspectBench Compiler[2](abc) [abc07], Soot[3] [soo07] and points-to and side-effect analyses in Soot. Then, we present the core impact analyses: state impact analyses to discover and record all state change caused by crosscutting; and computation impact analyses to classify computation impacts caused by applying advices into addition, elimination, must-substitution, may-substitution and invariant impacts.

3.1 Analyses Workbench

Our analyses are implemented in the back-end of the AspectBench Compiler - an alternative AspectJ compiler, which is developed by a joint team from Oxford, McGill and BRICS universities. The AspectBench Compiler itself is built on two established frameworks, and the back-end of abc is the Soot framework for Java analysis and optimization [ACH+05]. Soot provides five intermediate representations, all of which are highly suitable for analyzing tasks [VRHS+99]. The abc compiler uses Jimple as its IR, which is a typed, three-address and stackless and is also the principle IR of Soot. Jimple abstracts almost two hundred Java bytecode instructions into fourteen kinds of statements. All working class files, Java and AspectJ source code are transformed into Jimple in abc to be woven, and abc supports multi-phase weaving. Therefore, all our analyses are implemented using the Jimple IR.

An overview of the structure of abc, and where our analyses fits in, is given in Figure 1. The top part of the figure shows the high-level structure of abc, with the front-end connected to the back-end via the Java AST and the AspectInfo data structure. Our work is inside the box labeled "advice weaving + postprocessing", which is shown in more detail in the middle part of the figure. The advice weaver in abc is structured as a shadow finder, followed by a matcher, followed by a weaver, which results in woven Jimple. The analysis phase comes after the woven Jimple has been created, and this is where our impact analysis fits in. This is an ideal location for our analysis because at this point all intertype and advice weaving has been done, all advice bodies have been translated into normal Java methods (represented in Jimple), and we also have all of the information about the original aspects. As indicated by the bottom part of the figure, our impact analysis is one of the analyses applied at this stage of abc. Some other abc analyses are used to optimize the weaving, and in those cases the code may be rewoven. However, our impact analysis only needs to run on the first pass of the weaver as it does one analysis and then produces the impact report.

Points-to analysis is a static program analysis intended to estimate the set of locations pointed-to by a reference variable [EGH94]. Soot provides Spark, which is a customizable framework for inter-procedural, flow-insensitive and context-insensitive points-to [LH03, Lho02]. We use Spark in its default configuration, which provides a field-sensitive analysis (disambiguates fields by allocation sites) and on-the-fly call graph construction (builds the call graph during the points-to analysis).

Sometimes we need side-effect information rather than raw points-to information. Soot also contains a side-effect tester that can report if a statement, including a method call, possibly writes to a field. We take advantage of the side effect analysis in some of our impact analyses.

3.2 Considerations

Before discussing our analyses, there are several prior considerations to be clarified.

Firstly, our analyses needs points-to and call-graph information which can only be built if the whole program compiles, so our analyses cannot analyze partial programs.

Secondly, although inter-type declaration of aspects may also cause impacts on base program, the impacts described in this paper are due to advice. However, since we are implementing our system in abc, inter-type declarations will have been properly woven in the code that we are analyzing and so indirect effects on advice

[2]http://www.aspectbench.org
[3]http://www.sable.mcgill.ca/soot/

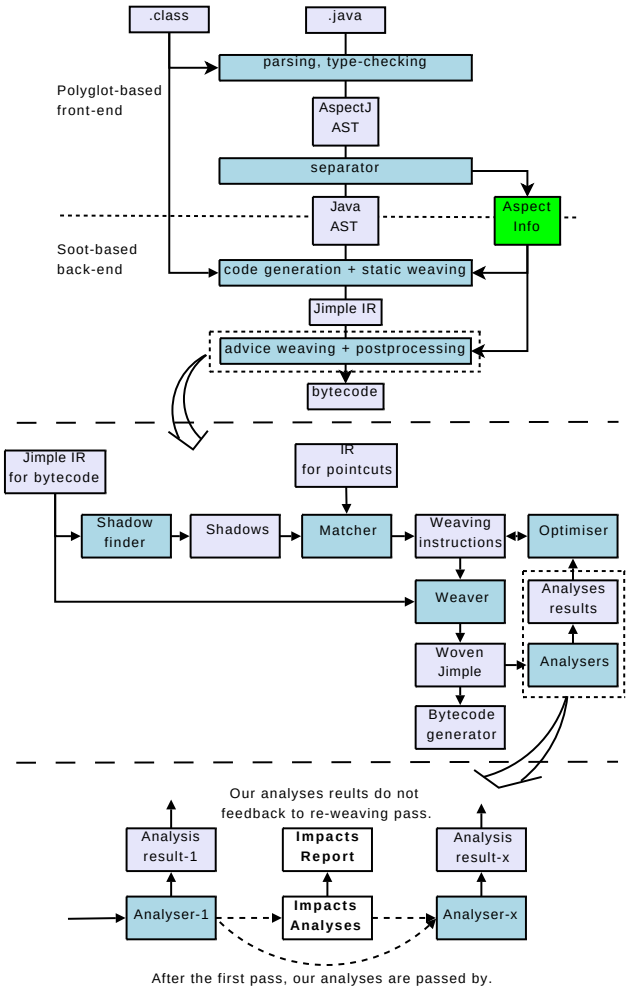

Figure 1: Impact analyses in abc, extended from [dM04]

impacts will be properly considered. In our future work we will also consider direct impacts of inter-type declarations, however the focus of this paper is impacts due to advice.

Last, but not least, an unchecked exception thrown in an advice may terminate the execution of the advice. We only take checked exceptions into account in our current analysis because every Java statement may throw unchecked exceptions, such as OutofMemoryError which leads to too many possible (but unlikely) exceptions. However, the programmer might be interested in certain unchecked exceptions, and we would like to extend our analysis so that programmers could provide us with a list of unchecked exception types of interest, and we will include impacts on all statements that throw those exceptions. This will require another interprocedural analysis, but is not difficult to do in our framework.

3.3 State Impact Analyses

Based on our classification of state impacts, our state impact analysis is designed to discover all field write accesses caused by advice, directly or indirectly. At conceptual level, the algorithm is presented in Figure 2.

```
for each advice ad in the application
{
   get corresponding Jimple body body
   for each statement stmt in body
   {
      if (LHS of stmt is FieldRef base.field) then
         if (base.field is static) then
            get class of base;
         end if
         if (base.field is instance) then
            get points-to set ptset of base;
            get types in ptset;
         end if
         record position;
      end if

      if (stmt contains method call method) then
         get Jimple body body of method;
         recursively call this analysis on body
         and record direct impacts caused by method
         as indirect impact of stmt;
      end if
   }
}
```

Figure 2: Pseudo-code for state impact analysis

As discussed earlier, our analyses focus on the granularity of advice, so we first acquire all advice information from abc after the first pass of weaving. Then, we iterate over each advice to perform the actual state impact analyses. As explained before, after the first pass of weaving, all advice bodies have been transformed to standard Java methods represented in Jimple IR. Thus, we can utilize the Soot analysis framework to perform our analysis.

3.3.1 Direct State Impacts

To discover direct field modifications, we need to check if the definition of an assignment statement is referring to a field in base classes. Since Jimple is a three-address IR, we actually only need check if the left hand side (LHS) of an assignment statement is referring to a field in base class. The field can be either a static field having the form *Class.field* or an instance field having the form *object.field*. In Soot, they are represented by two interfaces; the former is called StaticFieldRef, and the latter is called InstanceFieldRef. Therefore, we just need to iterate through all statements and check if the LHS is StaticFieldRef or an InstanceFieldRef. Moreover, since we are trying to find all possible field modifications, no matter whether the modification

happens in a branch path or not, we consider the modification as may-happening and as having direct state impacts.

Beside knowing where the impact happens, we need to know what happens, i.e. we need to know which field of which class is modified. The field name can be easily acquired by querying the FieldRef objects. For class information, in the case of a StaticFieldRef, we can easily get its declaring class. However, in the case of an InstanceFieldRef, the base object of the field may point-to objects with different types; thus, we need points-to analysis to determine the set of memory locations that the base object may point to, and then we can get the set of base classes that the field may belong to. In bank example, the points-to analysis will tell us that the actual type of the object in "account.lastVisit = new Date();"(Listing 4, line 10) in the bf before advice is CheckingAccount since we instantiated a CheckingAccount in Bank (Listing 3, line 7, 8).

All the information regarding where the impact happens and what is the impact is recorded and reported.

3.3.2 Indirect State Impacts

As stated in our classification, an advice can also cause indirect state impacts by calling methods which modify fields of base classes; thus, our analysis also checks indirect state impacts.

To modify fields indirectly, a statement inside an advice must call a method that modifies fields directly or indirectly. But, the method being called may call methods that modify fields too, so we check direct state impacts of all methods being called transitively. Moreover, call cycles, due to recursion, in the transitive call graph are resolved by maintaining a call stack and checking if a method is in the stack before entering and analyzing it. If the inspected advice is an around advice, it may call a special method - *proceed*. In our state impact analysis, we exclude *proceed* from the transitive call graph because *proceed* refers to computation in the base program and not new computation added by the advice. Although some *proceed* calls can produce different result with the computation in the base program, we have specific *proceed* analysis in our computation impact analyses; therefore, we exclude it here.

For all direct state impacts analyzed during traverse methods called transitively, we recorded them as evidence to support our indirect state impacts. In the same manner as with direct state impacts, we record both where and what happens for each piece of evidence. Moreover, to give an overall view of the indirect state impacts to programmer, we aggregate all points-to sets recorded when traversing the transitive call graph before outputting the report, grouped by field, by calculating the union of these points-to sets.

3.4 Computation Impact Analyses

Our computation impacts analysis is also designed on the granularity of advice and executed after the first weaving of abc. It classifies advice into addition, elimination, must-, may-substitution and invariant based on the kind of advice and *exact-proceed* analysis result. At the conceptual level, our algorithm is presented in Figure 3.

3.4.1 Exact-proceed analysis

As discussed in Section 2.3, our computation impact classification relies on the information regarding exact-proceed statements. Thus, to check if a proceed is an *exact-proceed*, we implement three intra-procedural analyses to check if those three conditions, *same arguments*, *same return*, and *no abrupt exception* are satisfied using the MustReachParamAnalysis, UnchangedReturnAnalysis and MayExceptionBeforeAnalysis, respectively. Although they are all intra-procedural analyses, the first two utilize inter-procedural analysis results, i.e. points-to and side-effect analysis results.

MustReachParamAnalysis This analysis collects all variables that have the same value of arguments, and its result is used to check the *same arguments* condition. It is implemented using the Soot forward flow analysis framework. For each program point the analysis computes a set of pairs, $(index, var)$. If pair (i, x) is in this set then this means that the variable x denotes exactly what the i^{th} parameter denoted at entry.

```
for each advice ad in the application
{
   if (ad is not around) then
      classify ad based on its kind;
   else
      if (empty computation) elimination;
      else
         exact-proceed analyses;

         if (exact one exact-proceed in every path
               and no other computation)
            invariant;
         else if (at least one exact-proceed in every path)
            addition;
         else if (no exact-proceed at all)
            must-substitution;
         else
            may-substitution;
      end if
   }
}
```

Figure 3: Pseudo-code for computation impact analysis

In particular, if the i^{th} parameter is a reference type, then (i, x) is in the set at program point p, only if x refers to the same object as the i^{th} parameter and there has been no write to a field of the object between the entry and program point p.

The analysis starts with the set $\{(1, param_1), \ldots, (n, param_n)\}$ at the entry point. New pairs are generated by copy statements of the form $lhs = rhs$ where the rhs variable in the input set. Pairs are killed in three ways. First, a statement of the form $lhs = rhs$ kills any pair associated with lhs. Second, we can also have killing due to aliases. For all statements of the form $a.f = b$, we kill any pair associated with a or any variable that may point-to to the same location as a (using the points-to analysis results). Third, if the rhs expression is a method call, it may write to objects. Thus, we use Soot's side-effect analysis as follows. For each pair in the input set, say (i, o) we check to see if there in an interference with the method call in rhs and any field of o. If there is an interference, then we kill the pair, because the state of the object may have been changed, and it may no longer denote the same value as at the entry point. Since this is a must analysis, we use intersection at control-flow merge points.

To illustrate this analysis, consider the example method in Listing 7. As indicated by the comment on line 2, at the entry point, set of must reaching parameters is $\{(1, a), (2, b)\}$. The copy statement at line 5 generates the pair $(1, a2)$. Thus, the **proceed** statement at line 7 does have the same arguments and in the correct order ($a2$ is the 1^{st} argument and and b is the 2^{nd} argument).

```
1  void around$0(A a, B b)
2    { // {(1,a),  (2,b)}
3      A a2, B b2;
4
5      a2 = a;
6      // {(1,a)  (1,a2),  (2,b)}
7      proceed(a2,b);
8
9      if (exp)
10       b2 = b; // {(1,a),  (1,a2),  (2,b),  (2,b2)}
11     else
12       b2 = new B(); // {(1,a), (1,a2), (2,b)}
13     // {(1,a),  (1,a2),  (2,b)}
14     proceed(a,b2);
15
```

```
16      foo(a); // foo writes to a field of a
17      // { (2,b) }
18      proceed(a,b);
19    }
```

Listing 7: MustReachParamAnalysis Example

Lines 9-14 illustrate a conditional and merge. On the true branch the pair $(2, b2)$ is generated, but not on the false branch. After the merge the intersection is computed as indicated in the comment on line 13, which does not include $(2, b2)$. Thus, the **proceed** at line 14 does not obey the same arguments property because $b2$ is not the same as the 2^{nd} parameter.

Line 16 illustrates the use of side-effects to kill. If we assume that the call to *foo* writes to a field of a, then the side-effect analysis will indicate an interference with the pairs $(1, a)$ and $(1, a2)$ and these pairs are therefore killed. Thus, the **proceed** at Line 17 does not obey the same arguments property because a is not the same as the 1^{st} parameter.

UnchangedReturnAnalysis In an **around** advice that returns a value, we must ensure that the value computed by a **proceed** actually reaches the return statement of the advice. We compute this in two steps. Given a statement of the form $x = proceed(\ldots)$, we first compute the set of statements that are reachable from this statement in control flow graph. For each such statement we check the same conditions as in the previous analysis, Namely we check if there is a direct write of the form $x = rhs$, a direct, or aliased write of the form $x.f = rhs$, or an interference on a method call. If no such writes occur, then we say that the value computed by the **proceed** reaches the end of the advice unchanged.

MayExceptionBeforeAnalysis This analysis checks if there are uncaught exceptions thrown before a given **proceed** statement, and its result is used to check the *no abrupt exception* condition. As discussed in Section 3.2, we only care about checked exceptions. Therefore, only statements in the same try-catch block with the given **proceed** statement, and executed before it, have the possibility to throw a exception that could cause the **proceed** statement to be bypassed.

Since an **around** advice cannot throw an exception itself, it follows that any checked exception thrown in the body of the advice must my caught be an enclosing try-catch block. Thus, we first check to see if the given **proceed** statement is surrounded by a try-catch block, if not, we conclude that there are no uncaught checked exceptions that can occur before the **proceed**. If there is a surrounding try-catch block, then we analyze the CFG starting at the first statement in the block. We compute the set of may reaching exceptions, using a standard forward analysis. We compute a set of exception types. We start at the beginning of the try-catch block with the empty set. A statement that possibly throws a checked exception generates a new element for the flow set (represented by the exception type). Exceptions may be thrown either by a direct throw statement, or by a call to a method that declares that exception in its signature.

Once the reaching exceptions are computed, we then check the flow set at the input of the **proceed** statement. If it is empty, then there are no abrupt exceptions, otherwise there may be an abrupt exception.

NumberExactProceedAnalysis With the above three analyses, we implement the NumberExactProceed-Analysis to count the number of exact-proceed in around advice.

First, we traverse all statements in the advice body looking for **proceed** calls. If the advice has void return type, which means the proceed call returns nothing, we apply MustReachParamAnalysis and the MayExceptionBeforeAnalysis on **proceed** call statements to check if the *same arguments* and *no abrupt exception* conditions are satisfied. If the advice has return type, we also apply the UnchangedReturnValue analysis. We collect all exact-proceed calls into a set. After that, we follow the CFG of the advice body to count if there is exactly one or more exact-proceed in all paths. At a merge point, we set our counter to the number coming from the path having less number of exact-proceed calls to guarantee our counter recording the number of exact-proceed on all paths.

3.4.2 Conclude Impact

Based on the results of the above analyses and the kind of advice, we conclude computation impact following rules below. For the issue of dead advice due to no matching, we rely on the abc warnings for all unmatched advices and we concentrate on the other tests.

Addition Impact We categorize all non-empty **before** and **after** advice as causing addition impacts. In addition, if an **around** advice has at least, but not exactly one, exact-proceed on every path and has additional computation other than the proceed calls, it has addition impact.

Elimination Impact If an **around** advice defines no computation in its body, it has elimination impact since the matched computation in the base program is totally removed.

Must-Substitution Impact If an **around** advice does not have exact-proceed on all paths and also defines new computation in its body, the matched computation is definitely replaced by computation defined in advice, so the **around** advice has must-substitution impact.

May-Substitution Impact If an **around** advice has at least one exact-proceed on some (but not all) paths and also defines new computation in its body, the matched computation is conditionally replaced by computation defined in advice, so the **around** advice has may-substitution impact.

Invariant Impact If there is exactly one exact-proceed on every path and no other computation is defined other than proceed-calls in an around advice, this **around** advice has invariant impact since it simply reproduces the matched computation in the base program.

Following the algorithm of our computation analyses, we revisit examples present in Section 2.1. In the bank example, our analyses should report that both the **before** and **after** advice (Listing 4, line 6 and 13) have addition impact because they are before/after advice and non-empty. The **around** advice (Listing 4, line 19) has exact one exact-proceed and extra computation, so our analyses should report addition impact also. In the source code repository example, the boolean **around** advice (Listing 6, line 5) has empty computation, so our analyses should report an elimination impact. The void **around** advice (Listing 6, line 10) has an exact-proceed in one path, but not in another path, so our analyses should report may-substitution.

4 Examples

To illustrate the kinds of reports we produce, we applied our system to three examples, the two examples introduced in Section 2.1, and another one given by [Rec05].

4.1 Bank

```
AccountAspect.aj:6,1-11:2 Advice: (in aspect bank.AccountAsp. . .
state impacts:
        bank\AccountAspect.aj:10,2-32
        direct state impacts:
        field [ lastVisit ] in [bank.CheckingAccount]
addition computation impact

AccountAspect.aj:13,1-17:2 Advice: (in aspect bank.AccountAsp. . .
state impacts:
        bank\AccountAspect.aj:16,2-15
        indirect state impacts:
        field [money] in [bank.CheckingAccount]
        evidence:
        bank\Account.java:16,20-39 field [money] in
          [bank.CheckingAccount]
```

addition computation impact

AccountAspect.aj:19,1−24:2 Advice: (in **aspect** bank.AccountAsp. . .
no state impacts.
addition computation impact

Listing 8: Report of analyzing bank

The report of our analyses on the bank example is shown in Listing 8. As expected and discussed in Section 3.4, our report shows that the **before** advice defined in Listing 4 between lines 6 and 11 has direct state impact of writing field lastVisit in CheckingAccount class, and the points-to analysis gives a precise estimation of the type of account(line 10). Moreover, the **after** advice in line 13-17 has indirect state impact on money field of CheckingAccount, and this indirect state impact is caused by the statement in Account.java at line 16 column 20-39, which is the money = money - FEE statement in Account.fee(). Therefore, our analyses reports not only exactly what happens, but also how it happens, so that programmers can use our analyses report to understand and analyze state impacts in very straightforward way. With the evidence report, programmers can trace back to the exact point where fields are written. In the glass and table example in Section 4.3, we will show how our state impact can help reveal bugs.

4.2 Source Code Repository

SourceCodeRepositoryAspect.aj:5,1−8:2 Advice: (in **aspect** repos. . .
no state impacts.
elimination computation impact

SourceCodeRepositoryAspect.aj:10,1−18:2 Advice: (in **aspect** rep. . .
no state impacts.
may−substitution computation impact

Listing 9: Report of analyzing source code repository

Listing 9 shows the report of analyzing source code repository example. As expected, our analyses classifies the **around** advice defined in SourceCodeRepositoryAspect.aj in line 5-8 has elimination computation impact, and the **around** advice in line 10-18 has may-substitution computation impact. Through our report, programmers can get a general view of what an advice is going to do if applying the advice, or use our conclusion to verify if the advice is implemented as expected. For example, if advice is designed to totally replace the computation in the base program; but, our analyses shows the advice has may-substitution computation impact, the programmer just needs to focus on checking if there is an exact-proceed on some paths. Therefore, our analyses can also reduce the difficulty of finding bugs in advice. Moreover, as discussed before, invariant computation impact itself sometimes also an indication of bugs.

4.3 Glass and Table

In this example, in the base program, a Table contains a Glass, and Table.move() calls Glass.move(), thus if the table moves, the glass also moves. Then, an aspect applies to the program. In this aspect, there is an **after** advice crosscuts execution of Glass.move(), and this advice call Table.move() to accomplish the purpose of moving tables if the glass moves. If we examine both the aspect and the base program, we can discover this advice causes a call cycle, which causes infinite loop. If running the woven program, it ends up with a StackOverflowError error.

GlassAspect.aj:6,1−10:2 Advice: (in **aspect** glass.GlassAspect) . . .
no state impacts.
addition computation impact

```
GlassAspect.aj:12,1-17:2 Advice: (in aspect glass.GlassAspect). . .
state impacts:
        glass\GlassAspect.aj:16,3-21
        indirect state impacts:
        field [x] in [glass.Table]
        field [x] in [glass.Glass]
        field [y] in [glass.Table]
        field [y] in [glass.Glass]
        evidence:
        glass\Table.java:19,2-9 field [y] in [glass.Table]
        glass\Table.java:18,2-9 field [x] in [glass.Table]
        glass\Glass.java:9,2-9 field [y] in [glass.Glass]
        glass\Glass.java:8,2-9 field [x] in [glass.Glass]
addition computation impact
```

Listing 10: Report of analyzing glass and table

Listing 10 shows the report of our analysis on glass and table program. We can see our report points out the second advice has state impacts both on Glass and Table. Programmers could utilize this information to conclude this advice does something more than expected. The original purpose of this advice is to move the table if the glass moves, but according to our report it also writes fields in Table class, this should not happen; thus, there are mistakes. Moreover, if the programmer follows our evidence and checks Table.java and Glass.java, he/she will find these statements are defined in Glass.move() and Table.move(), so this advice actually transitively call both Glass.move() and Table.move(). Then, the programmer should very easily conclude there is a call-cycle. For this simple example, switching back and forth between different source files perhaps can be handled, but in larger applications the programmer needs more direction such as provided by our impact report.

Clearly our textual reports could also be integrated into a GUI, and one of our next goals it to integrate our report information into a GUI.

4.4 Classification of aspects or advices

We can also use our impact analysis to define higher-level classifications. For example, we could support a classification of *data-pure* that would indicate when an advice had no state impacts, or *data-pure on class C* which would indicate no state impacts on a given class *C*. With this classification, the **around** advice (Listing 4, line 19) in bank example and the two **around** in source code repository example are all *data-pure* advice since they have no state impacts.

We could also support the idea of *computation-pure*, which would indicate that the aspects did not remove any base program computation. Thus, advice that does not have an *elimination*, *may-substitution* or *must-substitution* impact is *computation-pure*. All advice declarations except the second **around** (Listing 6, line 10) in source code repository example are *computation-pure*.

We could also define and advice as being *pure* it it is both *data-pure* and *computation-pure*.

In addition, our analyses result can also be used to classify aspects or advice under another classification system. Based on our analyses, a *harmless advice* [DW06] is an advice without *state impacts*, *may-* and *must-substitution*, *elimination computation impacts*; in fact, it corresponds to our *pure advice*. Advice definitions having state impacts would cover advice having *actuation* and *interference* interaction in Rinard's classification.

5 Related Work

The problem of helping the programmer understand the impacts of aspects is not a new one and there has been some interesting previous work in this area.

Starting at the syntactic/IDE level, the eclipse plug-in AJDT[4] [ajd07] provides visualizations to indicate shadows where advice applies in the base program, thus providing some cues for the programmer as to places in the base program which might be affected. However, to discover the actual impact of aspect on the base program, the programmer has to manually review the source code and possibly has to frequently switch between base program source and aspect source. A key difference in our approach is that we are using static analysis to find more detailed information for programmers. However, the idea of visualizing the information in AJDT is very interesting and we hope to leverage some of the visualization tools in AJDT in our next step to expose our more detailed information to programmers.

Strözer introduced an aspect analysis, and although he stated that aspect analysis required data flow analysis, but he had no infrastructure tools available [Str03], thus he proposed using trace analysis to fulfill the impact analysis. His approach relies on comparison of two traces of the program without and with applied aspect for a single test. Then, by identifying patterns of differences, the impact of an aspect can be observed [SKB03]. However, firstly, his approach heavily depended on the quality of test case; secondly, the report can only vaguely describe the impact at the level of an aspect. Since our infrastructure does support data flow analysis, we were able to actually implement static impact checking.

Recebli analyzed different ways aspects can break encapsulation and proposed the purity aspect language feature to AspectJ [Rec05]. Through this feature, a programmer can declare an aspect is *pure on* a specified set of classes by promising that the aspect will not change the behavior of the set of classes. Moreover, he presented an implementation of the proposed purity annotation as an abc extension and used static analysis to verify purity. Our approach is more focused on the non-pure impacts and on ways of categorizing and approximating those impacts. Our static analysis is also somewhat more detailed as we also take into consideration side effects of method calls when analyzing *proceed*.

Dantas introduced the concept of *harmless advice*, which works like ordinary aspect-oriented advice but is designed to obey a weak non-interference property, i.e. it may change the base program's termination behavior and use I/O, but it does not influence the final result of the mainline program. In order to detect and enforce harmlessness, they defined a novel type and effect system related to information-flow type systems. They also presented an implementation of the language [DW06]. However, their work was done at a very abstract level and is hard to integrate it directly into AspectJ. Our approach is more focused on developing classifications and associated analyses that have been integrated into an AspectJ compiler.

Perhaps the most directly related work was from MIT by Rinard et. al. [RSB04], which also was designed to work on AspectJ and used static analysis. There are clearly similarities between the two approaches as both seek to summarize state impacts and computation impacts (although these were not the terms used in the MIT paper). From a conceptual point of view the approaches differ in the manner in which the impacts are abstracted. For example, in the MIT approach, state impacts are expressed as interferences between fields accessed by a base program method and fields accessed by an advice, whereas our approach takes a more advice-centric approach, and we report all fields of the base program written by an advice. We believe that our approach will lead to more direct reports and is more easily integrated into an IDE. From an implementation point of view there are also similarities and differences. Both systems are built on Java bytecode frameworks which support points-to analysis. The MIT prototype used the byteocde produced by ajc as input, and used the MIT Flex compiler infrastructure for the static analyses. Their implementation was limited to method call and method execution join points, perhaps because of the loose coupling between ajc and Flex. Our approach is implemented directly in the abc compiler and so we have access to all the necessary information to handle all kinds of join points. Further, our implementation will be merged into the main stream abc release and will be available for others to build upon.

6 Conclusions and Future Work

As AOP, especially AspectJ, is becoming increasing popular, we believe that tools can help programmers understand the complicated interaction between aspects and base programs. The design of tools that can

[4] http://www.eclipse.org/ajdt/

compute both useful and accurate information presents many interesting and challenging problems and the availability of such tools should help increase adoption of AOP.

In this paper, we presented different ways that advice can interfere with the state and computation of a base program and proposed a concise classification of impacts caused by advice crosscutting the base program. We classified impacts as *state* and *computation impacts*, and further classified *state* impacts into *direct state impact* and *indirect state impact*; and classified *computation impacts* as *addition*, *elimination*, *must-substitution*, *may-substitution* and *invariant*.

Based on this classification, we implemented static impact analyses in the abc compiler to analyze all kinds of advice. By using the points-to analysis and side-effect analysis supported by Soot, our impact analyses system can give precise estimations of impacts. Our approach also produces an informative analysis report. In the report, we not only report the impact information, but also report the causes of impacts, so we can guide the programmer to understand the key impacts of aspects on their program.

We feel that this initial work shows that our approach has promise, and we plan to continue on with this work, extending it in the following ways.

Although our textual reports are quite concise, modern developers appreciate analysis results that are integrated into an IDE. Thus, we plan to integrate our toolkit into Eclipse and produce reports that will allow the programmer to both view the impacts in a more graphical manner and allow programmers to navigate directly to the parts of the source code involved in the impacts.

There are also several extensions to the analyses that we would like to undertake. First, it would be interesting to add the analysis for programmer-specified unchecked exceptions, because in practice, unchecked exceptions are used more frequently than checked exceptions. We would also like to experiment with different points-to analyses, since the precision of our impact analyses heavily depends on the points-to result. There are two interesting context-sensitive analyses available for Soot now, the Paddle framework [LH06, Lho06], and the demand-driven analysis of Sridharan and Bodik [SB06]. It should be simple to integrate these into our approach, and a study of the effect of points-to precision on the quality of impact reports would be very interesting.

Finally, to handle all kinds of aspects available in AspectJ, we wish to add intertype impacts to our system. We have an initial design of kinds of impacts we would like to expose and it should be possible to integrate these smoothly into our framework.

Acknowledgements

This research has been supported by Le Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[abc07] abc - AspectBench Compiler. `http://www.aspectbench.org`, 2007.

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.

[ajd07] AspectJ Development Tools (AJDT). `www.eclipse.org/ajdt`, 2007.

[dM04] Oege de Moor. abc: an implementation of AspectJ. Seminar at the Computer Laboratory, Cambridge, United Kingdom, `http://abc.comlab.ox.ac.uk/documents/dec8.pdf`, December 2004.

[DW06] Daniel S. Dantas and David Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM Press.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[LH06] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference*, volume 3923 of *LNCS*, pages 47–64, Vienna, March 2006. Springer.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[Rec05] Elcin Recebli. Pure aspects. Master's thesis, Oxford University, September 2005.

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM Press.

[SKB03] Maximilian Störzer, Jens Krinke, and Silvia Breu. Trace analysis for aspect application. In *Analysis of Aspect-Oriented Software (AAOS)*, 2003.

[soo07] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`, 2007.

[Stö03] Maximilian Störzer. Analytical problems and AspectJ. AOSD workshop, 2003. Talk.

[Str03] Maximilian Störzer. Analysis of AspectJ programs. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, 2003.

[SW07] Martin Sulzmann and Meng Wang. Aspect-oriented programming with type classes. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, pages 65–74, New York, NY, USA, 2007. ACM Press.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.