Impact Analysis and Visualization Toolkit for Static Crosscutting in AspectJ

Dehua Zhang, Ekwa Duala-Ekoko, and Laurie Hendren
School of Computer Science, McGill University,
Montreal, Quebec, Canada
{dehua.zhang@mail, ekwa@cs, hendren@cs}.mcgill.ca

Abstract

Understanding aspect-oriented systems, without appropriate tool support, is a difficult and a recognized problem in the research community. Surprisingly, little has been done to help developers understand the impact of the static crosscutting constructs of AspectJ on base programs. Questions of interest to developers such as: which statements in a base program are affected by a given inter-type declaration, or how has the behavior of the affected statements been modified, are still outstanding. This paper presents analysis techniques for inferring the impact of the static crosscutting constructs of AspectJ on base programs, and tools for visualizing the results of the analysis; thus improving the comprehension of AspectJ systems and guarding against unintended modifications. Our analyses are implemented as extensions to the AspectBench compiler, and integrated in the Eclipse IDE as a plugin. We present experiments on several open source systems to investigate the effectiveness and suitability of our analysis techniques and tools.

1 Introduction

Aspect-oriented programming (*AOP*) languages introduced constructs for modularizing crosscutting concerns in units called *aspects* [5, 8, 12], and support for combining aspects with object-oriented programs to produce a final system. AspectJ [8, 10], a widely used Java implementation of AOP, provides both dynamic constructs, called *advice*, and static constructs, called *inter-type declarations*. Advice modify the behavior of the classes they crosscut, while inter-type declarations modify their static structure. Although powerful, the use of these constructs raises new challenges to program comprehension [11]. In particular, the oblivious nature of these constructs combined with the polymorphic features of object-oriented languages makes reasoning about the structure and behavior of AspectJ systems a lot more difficult without appropriate tool support. Simple modifications such as introducing a new method to a class from an aspect may impact untouched areas of the base application in unintended ways. Failure to immediately identify and resolve such unintended impacts may compromise the structure or state of the base application.

Several tools for generating and visualizing the impact of AspectJ advice on a base program have been proposed in the literature [6, 13, 14, 17]. For instance, given a *before* advice (i.e., a piece if code to run before a specified point in the execution of a base program), the AspectJ Development Tools (AJDT) of Eclipse [3, 6] will identify all the locations in the base program affected by the advice — helping developers to reason about the behavior of their application in the presence of aspects. Surprisingly, little attention has been given to the impact of inter-type declarations on base programs. For instance, a single inter-type declaration, such as `declare parents: classX extends classY` which makes `classY` the new parent of `classX`, may significantly modify the inheritance hierarchy of several other classes in a base program, even altering the method dispatch patterns of call sites or even field references, sometimes adversely. However, not even AJDT — the state-of-the-art for understanding AspectJ systems, supports reasoning about the impact of such static constructs on base programs. Consequently, major questions of interest to developers such as: *which locations in a base program are affected by a static crosscutting construct*, or *how have references to methods or fields in the base program been impacted by static crosscutting constructs*, must be manually determined — a procedure made tedious and error prone because of the oblivious nature of aspects.

To address these challenges, we studied the interaction patterns between the static crosscutting constructs of AspectJ and base programs, and categorized aspects into three groups — based on their impact on the state (i.e., fields) and behavior (i.e., methods) of a base program. We implemented a set of static analysis techniques to automatically identify these interaction patterns, and provide an Eclipse plugin, called *ITDVisualiser*,[1] for visualizing the results of the analyses and for reasoning about the impact of static

[1]ITDVisualiser is available at: www.cs.mcgill.ca/∼eduala/itd-tool/

constructs on base programs. Our approach not only identifies the fields and methods in the base program affected by aspects, but also provides an explanation of how each statement was impacted. Tools such as *ITDVisualiser* are necessary not only for program comprehension, but also as a first line of defense against the misuse of static crosscutting constructs, since developers would immediately identify unintended modifications and make adjustments where necessary. We make the following specific contributions:

- We provide a classification of the interaction patterns between the static crosscutting constructs of AspectJ and base programs.
- We present analysis techniques capable of identifying the interaction patterns automatically, and tools that implement our analyses with visualization and reasoning support of the results in an IDE.
- We provide an evaluation of the effectiveness and suitability of the analysis techniques and tools using five open source AspectJ systems.

This paper is organized as follows. We discuss the challenges developers face when reasoning about the impact of static constructs on base programs in Section 2. Our analysis techniques for identifying the impact of aspects on base programs is presented in Section 3, and a tool implementation of the analysis is presented in Section 4. We present an evaluation of our tool in Section 5, discuss related work in Section 6, and conclude the paper in Section 7.

2 Motivation

AspectJ provides several constructs for modifying the structure of existing Java applications. For our purpose, we focus on the construct for modifying the inheritance hierarchy (i.e., `declare parents`), and those for introducing new methods or fields to Java classes (i.e., `inter-type declarations`) since the use of these could modify the state or behavior of the base application. Understanding the extent of the impact of these constructs on a base program is difficult, and their use, without appropriate tool support, risk the introduction of unintended behavior in a system. We use the sample application in Figure 1 to highlight some of the challenges these constructs pose to program comprehension. Our application has five classes: `Shape`, `Line`, `Rectangle`, `Square`, and `Main` – the driver of our application. We have placed all the *call sites* — the locations from which methods are called — within the `main` method of the class `Main` for convenience; in practice these would be anywhere in the application.

```
abstract class Shape{
  protected int xPos, yPos;
  protected Color myColor = Color.blue;

  public void setColor(Color color){
    myColor = color;
  }

  public void move(int x, int y){
        //move shape & update GUI
  }

  public abstract void draw();
}

class Line extends Shape{
  public void draw(){
    // draw line
  }

  public void move(int x, int y){
   // move line & update GUI
  }
}

class Rectangle extends Shape{
  public void draw(){
    // draw rectangle
  }
}

class Square extends Shape{
  public void draw(){
    // draw rectangle
  }
}

class Main{
  static void main(String[] args){
  Square square = new Square();
  Rectangle rect = new Rectangle();
  SystemColor color = getSystemColor();
  square.setColor(color);
  square.draw(); rect.draw();
  square.move(40,80);
  rect.move(40,60);
 }

  static SystemColor getSystemColor(){
    // get default system color
  }

}
```

Figure 1. Sample Base Application

IDEs such as Eclipse[2] provide features for navigation and searches, including navigating to the declaration of a selected element, and support for searching all the references to a selected element. This makes reasoning about the structure and behavior of our application, without static crosscutting aspects, not too difficult. In our case, a developer would infer that the call sites `rect.move(40,60)` and `square.move(40,80)` would be dispatched to `Shape.move(int,int)` since `move(int,int)` is not redefined in either `Rectangle` or `Square`. Similarly, the call `square.setColor(color)` would be dispatched to `Shape.setColor(Color)`, and all references to the field `myColor` in our application refers to `Shape.myColor`. Reasoning about the behavior of these same elements in the presence of static crosscutting aspects is not as straight forward. Specifically, the oblivious nature of aspects, the presence of multiple aspects, and the subtyping feature of object-oriented languages, Java in our case, makes program comprehension particularly difficult.

Reasoning in the Presence of Obliviousness: Aspects by nature are oblivious — that is, base programs are completely unaware of their existence. Thus, developers can no longer rely on the traditional navigation and search features of Eclipse to understand the behavior or structure of a system.[3] For instance, consider the aspect in Figure 2(a), `IntroduceToRectangle`, that introduces the method `move(int,int)` and the field `myColor` of type `Color` to the class `Rectangle`, as the only aspect in the system. This simple modification has changed the method dispatch of the call site `rect.move(40,60)` from `Shape.move(int,int)` to `Rectangle.move(int,int)`, and all references to `myColor` in `Rectangle` from `Shape.myColor` to `Rectangle.myColor`. However, none of the existing navigation and search features of Eclipse can identify these major changes; they still report the call `rect.move(40,60)` as referring to `Shape.move(int,int)`, and `myColor` as referring to `Shape.myColor`. The base program is completely unaware that the class `Rectangle` now has `move(int,int)` and `myColor` as part of its interface. As noted by Kiczales and Mezini [9, p3], one "...cannot know the complete interfaces of modules in a [AspectJ] system until we have a complete system configuration...". Such incomplete knowledge of the aspects in a system by the base program makes program comprehension error-prone and difficult.

[2]www.eclipse.org

[3]The AspectJ compiler is implemented as an extension of the Eclipse Java compiler, but the Java model from which the Eclipse navigation and search features are built is not extended. As such, the navigation and search features of Eclipse cannot "see" modifications made by static crosscutting constructs [3].

```
aspect IntroduceToRectangle{
 public void Rectangle.move(int x,int y){
 // move rectangle & update GUI
 }

 public Color Rectangle.myColor = Color.yellow;
}
```
(a) *Introduce a public method move(int,int) and a field myColor of type Color to Rectangle.*

```
aspect ModifyHierarchyOfSquare{
  declare parents: Square extends Rectangle;
}
```
(b) *Modify the parent of Square from Shape to Rectangle*

Figure 2. Sample AspectJ aspects I

Reasoning in the Presence of Multiple Aspects: Even with complete knowledge of the aspects in a system, reasoning about call sites or field references in a base program is still challenging since the actual targets of an element may be due to the impact of a combination of two or more static crosscutting aspects. Let us now assume that our system has both aspects defined in Figure 2. Reasoning in isolation (i.e., one aspect at a time) about the impact of these aspects on our base application would produce an incomplete and erroneous picture in certain cases. For instance, if we consider only the `ModifyHierarchyOfSquare` aspect that modifies the parent of `Square` from `Shape` to `Rectangle`, we would conclude that the call `square.move(40,80)` will be dispatched to `Shape.move(int,int)` since `Square` now extends `Rectangle` and, based on our isolated knowledge, `Rectangle` does not redefine the method `move(int,int)`. This view is obviously erroneous since `Rectangle` now redefines `move(int,int)` because of the `IntroduceToRectangle` aspect, thus, the call `square.move(40,80)` is actually dispatched to `Rectangle.move(int,int)`. As demonstrated by our trivial example, the impact on a base program may be from a combination of multiple static crosscutting aspects. Reasoning about such impacts manually is cognitively challenging, and highlights the need for tools that reflect the complete picture.

```
aspect IntroduceToShape{
 public void Shape.setColor(SystemColor color)
 {
  //do something
 }
}
```

Figure 3. Sample AspectJ aspect II

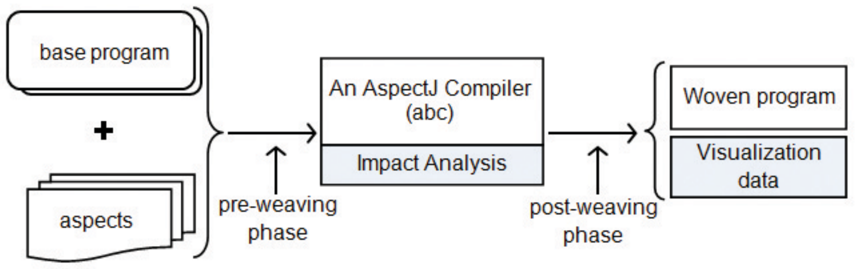


Figure 4. An overview of the AOP approach. An AspectJ compiler combines the base program with the aspects to produce a final system (woven program). Our impact analysis, implemented as an extention to the AspectBench Compiler (abc), outputs visualization data made available through the ITDVisualiser plugin views.

Reasoning in the Presence of Subtyping: Overloading methods based on argument types may break code in classes untouched by a modification task. This occurs when the arguments of an introduced method are subtypes of the respective arguments of the overloaded method, and also more specific to the arguments of call sites in a base program. For instance, consider the aspect in Figure 3 which introduces the method `setColor(SystemColor color)` to the class `Shape`. Observe that the method `Shape.setColor(SystemColor color)` overloads the method `Shape.setColor(Color color)`, and that the class `java.awt.SystemColor` is a subtype of `java.awt.Color`. This modification would change the dispatch of the call site `square.setColor(color)`, in class `Main`, from `Shape.setColor(Color color)` to `Shape.setColor(SystemColor color)` since the argument `color`, of type `SystemColor`, is more specific to `setColor(SystemColor color)`. Identifying such subtle impacts, without the right tool, is almost impossible since the root cause is not even part of the base application.

Our Solution: Understanding the impact of static crosscutting aspects is far from easy as demonstrated by our trivial sample application. To address the identfied challenges, we developed a tool that automatically classifies static crosscutting aspects based on their impact on the state and behavior on the base program. We have identified the following interaction patterns:

- **Lookup impact:** this interaction pattern represents static constructs that modify the method dispatch pattern of one or more call sites in a base program.
- **Shadowing impact:** this interaction pattern represents static constructs that modify the references to one or more fields in a base program.
- **Orthogonal:** this interaction pattern represents static constructs that have no impact on either the field references or method dispatch patterns of statements in a base program.

Some aspects can have both lookup and shadowing impacts on statements in the base program. Given an aspectJ application (base program and aspects, see Figure 4), our tool automatically classifies the aspects, and provide support for visualizing and reasoning about their impact on the base program. This way developers can have a complete picture, and better comprehend the current structure and behavior of the application. We present the analyses techniques and algorithms of our approach in the next section of this paper.

3 Analysis Techniques

To identify the interaction patterns between static constructs and base programs, and to determine changes to field references or method dispatch patterns, our analysis needs to know the structure of the application both before and after the aspects have been woven into a base program. To obtain this information, we implemented our analysis as part of the *AspectBench Compiler* (abc) [2]. abc is a complete implementation of AspectJ, with support for extending both the *pre-weaving* and *post-weaving* phases of the aspect weaving process (see Figure 4), and for obtaining information about the aspects in a system. The pre-weaving phase gives the structure of the code before the aspects are applied. The post-weaving phase provides us with complete knowledge of the structural modifications caused by the aspects, addressing the “Reasoning in the Presence of Obliviousness” and “Reasoning in the Presence of Multiple Aspects” problems identified above. Our analysis techniques could be regarded as a *diff* between the post-weaving structure (i.e., *woven code*) and the pre-weaving structure (i.e., *original code*) of a system, with emphasis on changes to field references and method dispatch patterns.

3.1 Lookup Impact Analysis

To cause a change to the method dispatch of a call site, a static construct must satisfy two requirements: first, the modification made by the construct must have a *polymorphic-impact* (i.e., overload or redefine) on one or more methods in the base program; secondly, the methods affected must be referenced by statements in the base program. Our lookup impact analysis technique is therefore a two step process:

Step 1 [Is there a polymorphic-impact?]: To answer this question, we first identify classes in the base program affected by either an inter-type method declaration *(IMD)*, or a declare parents *(DP)* construct of AspectJ. We obtain the list of static constructs from abc, and perform *Class Hierarchy Analysis* [15] on the woven code to identify the set of affected classes (henceforth referred to as *shortlisted-classes*) for each static crosscutting construct. The set of shortlisted-classes for an IMD construct is made up of the `target-class` of the introduction and all the subtypes of the `target-class`. For instance, the set of shortlisted-classes for the IMD, `Rectangle.move(int x,int y){...}`, from Figure 2, is {Rectangle, Square} since Rectangle is the target-class and Square is a subtype of Rectangle because of the `ModifyHierarchyOfSquare` aspect. The set of shortlisted-classes of a DP construct (such as `declare parents: TypePattern extends Type`) is made up of all the classes in the base application whose name matches `TypePattern` and their corresponding subtypes.

Given the shortlisted-classes for each construct, we can now verify cases of polymorphic-impact. For an IMD, such as `ReturnType OnType.methodName(`P_1`,`P_2`, ...)`, the newly introduced method, `methodName(`P_1`,`P_2`, ...)`, has a polymorphic-impact on the base code if it overloads a method in class `OnType` based on the argument types (P_1,P_2, ...), or redefines an inherited method. If the introduced method neither overload nor redefines an existing method, we safely conclude that the IMD has no polymorphic-impact on the base program, and thus, it is classified as being orthogonal to the base code.

Observe that the overloaded or redefined method will serve as the *previous* method dispatch of calls to `methodName(`P_1`,`P_2`, ...)` with run-time receiver type `OnType`. In certain cases, the introduced method may overload or redefine more than one method. In such cases, we identify the previous method dispatch of the overloaded or redefined method using the *most-specific* method concept of the Java Language Specification (JLS). According to the JLS, a method m_i is considered most-specific to the newly introduced method `methodName(`P_1`,`P_2`, ...)` than another method m_j if the argument types of m_i are subtypes of the respective argument types (P_1,P_2, ...) of the introduced method.

The DP construct permits a class to extend either its siblings or its siblings sub-classes. This restriction implies a class can only be moved down the hierarchy, and thus, may only inherit new methods, but cannot lose methods. If a class `X` extends a sibling or a sibling's sub-class `Y`, the newly inherited methods in `X` is the set of the non-private methods in the class `Y`, both declared and inherited. We treat each newly inherited method by the class `X` as an IMD, and proceed as above to verify cases of polymorphic-impacts. The output of *step 1* is the *polymorphic-impact information* for each construct that details the impacted classes and the signature of the polymorphically impacted methods.

Step 2 [Which call sites in the base program are affected, and how?]: Not every polymorphic-impacted class would immediately result in an impact in the base program. This is because to cause changes to method dispatch, and thus impact the base program, the method overloaded or redefined by a construct must be referenced at one or more call sites in the program. The second step of the lookup impact analysis examines call sites in the program to identify changes to method dispatch patterns.

The polymorphic nature of object-oriented languages, and Java in particular, makes it difficult to resolve the run-time target of call sites at compile time. Notwithstanding, static analysis techniques can be used to generate a *precise call graph* good enough for our purpose — inferring a precise set of potential targets for a given call site. For our purpose, a call graph is a set of nodes and edges with one node for each method, m_i, in the application, one node for each call site, c_i, in the application, and an edge from c_i to m_i, if c_i may call m_i. We used the `Soot` framework [16] to generate the call graph. The call graph construction algorithm begins with a *conservative call graph* generated using *Class Hierarchy Analysis*. It then uses *Rapid Type Analysis* [15] and *Points-to Analysis* [7] to improve the precision of the call graph by eliminating the unlikely targets of a call site.

Once the polymorphic-impact information and a precise call graph of the application are known, we then proceed to determine the method dispatch changes of the affected call sites. Assuming the declared type of `object` is `OnType`, the call site `object.methodName(param1,...)` is considered affected if `OnType.methodName(param1,...)` is in the set of the polymorphic-impacted methods. The outgoing edge(s) of the affected call sites in the precise call graph gives us the new method dispatch. A static construct in the polymorphic-impact set that affects one or more call sites is classified as having a lookup impact; otherwise, we consider the construct as being orthogonal to the base program.

3.2 Shadowing Impact Analysis

For a shadowing impact to occur in a class C, the modifications made by static constructs must introduce a field with the same *name* and *type* as a field inherited by C from a superclass. Two of the static constructs of AspectJ are capable of causing shadowing impact: the inter-type field declaration (IFD) and the declare-parents-extends declaration (DPED). Our analysis examines the classes affected by these constructs to confirm the presence or absence of shadowing impacts.

Does an IFD cause a shadowing impact? An IFD has the form "`Type TargetClass.foo`"; that is, it introduces a new field named `foo` of type `Type` into the class `TargetClass`. To determine cases of shadowing impact, our analysis examines the class hierarchy of `TargetClass` in the original code, looking for a superclass with a field named `foo` of type `Type`. If no such superclass is found, our analysis classifies the IFD as being orthogonal to the base program. If, on the other hand, we encounter a superclass, `SuperC`, of the `TargetClass` with a field named `foo`, and if the `TargetClass` has a reference to `foo`, we classify the IFD as having a shadowing impact on the base program. Our analysis reports `SuperC.foo` as the *previous* target to references to `foo` within the `TargetClass`, and `TargetClass.foo` as the *current* target.

The shadowing impact on the `TargetClass` may also be propagated to some of its subclasses; that is, to subclasses with references to the field `foo`, but that do not redefine `foo`. Our analysis further examines the subclasses of `TargetClass`, and generates a similar report for each affected subclass.

Does a DPED cause a shadowing impact? A DPED construct, `declare parents: TargetClass extends NewParent`, modifies the immediate parent of the `TargetClass` from a previous class, `OldParent`, to a new class, `NewParent`. Since DPED permit `TargetClass` to only extend a sibling or its siblings' sub-classes, `TargetClass` can only inherit, but not lose, fields. To cause a shadowing impact on a field named `foo`, referenced within `TargetClass`, the following two conditions must be satisfied:

1. An existing, but older, superclass of the `TargetClass` — that is, a superclass of the `TargetClass` before the inter-type extends declarations are woven into the base program — has a field named `foo`, inherited and referenced within the `TargetClass`.

2. A new superclass of the `TargetClass` — that is, a superclass newly introduced after the inter-type extends declarations are woven into the base program — also has a field named `foo`, of the same type as in (1). Since, the `TargetClass` was moved down the hierarchy, the `foo` in the new super class will shadow the `foo` in the older superclass.

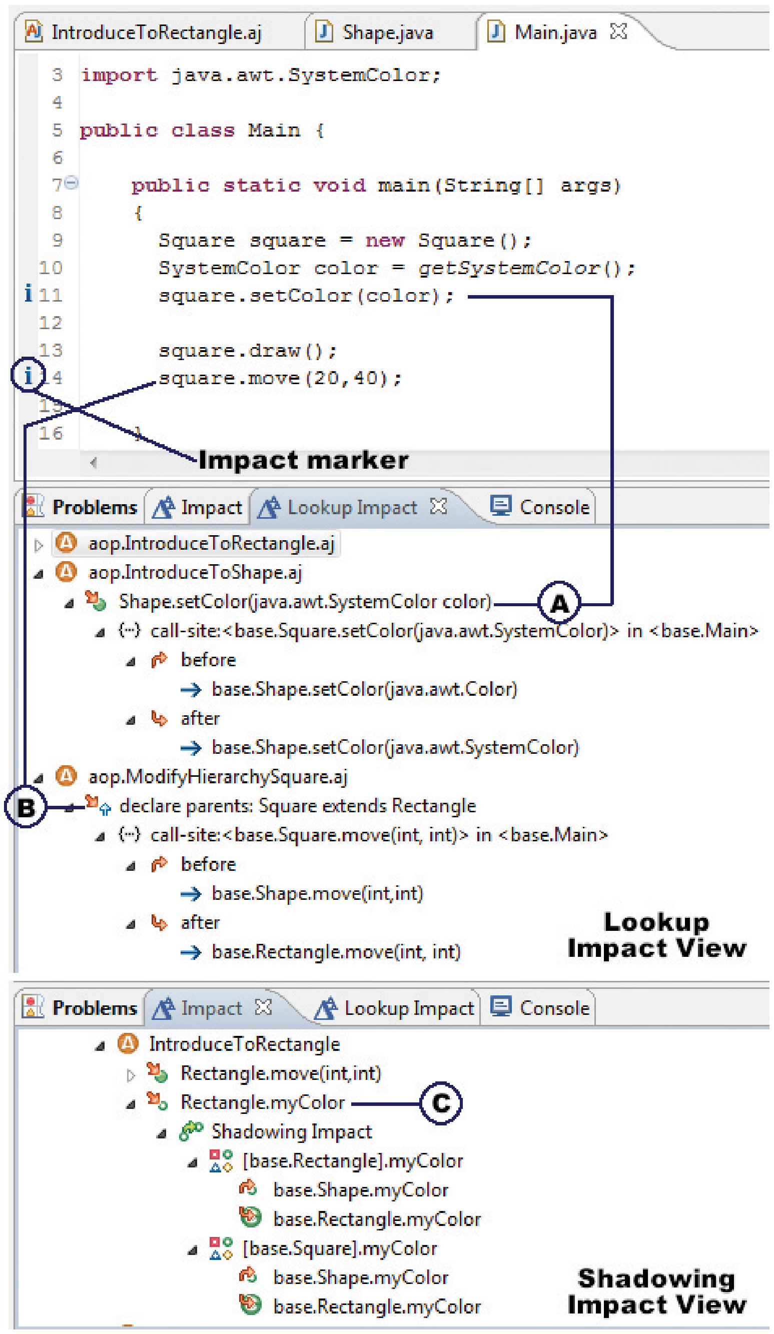
To verify these conditions, our analysis begins by first identifying the fields newly inherited by the `TargetClass`. These include all non-private fields declared in classes along the path between the old parent (exclusive) and the new parent (inclusive) of `TargetClass`. Our use of abc enables us to easily obtain both the old and new parent by extending the *pre-weaving* and *post-weaving* phases of abc.

We then traverse the path from the old parent to the new parent recording all fields declared in classes along the path and inherited by the `TargetClass`. In case a class down in the path declares a field having the same name and type as a field declared in a class above it, and both fields are inheritable by the target class, only the field declared in the class further down in the path is recorded. Each recorded field is then treated as a newly introduced field on the `TargetClass`, and our analysis proceeds similarly as that of the IFD to determine the presence or absence of shadowing impact.

4 Visualization Tools

We have implemented the lookup and shadowing impact analysis as an Eclipse plug-in called ITDVisualiser. To identify and reason about the impact of static constructs on a base program, such as the program from Figure 1, a developer would right-click on the AspectJ project in the Package Explore view of Eclipse, and select the ITDVisualiser tool from the context menu. ITDVisualiser will prompt the developer for the main class of the application (required to construct the precise call graph) and the source code directory, and then runs the lookup and shadowing impact analysis. The ITDVisualiser tool makes three contributions to facilitate reasoning about the behavior of systems with static constructs: the *impact marker* identifies statements whose behavior have been modified by one or more static constructs, the *views* display the results of the lookup and shadowing analysis, and the *bi-directional navigation* feature provides support for navigation from the affected source code to the views, and vice versa (see Figure 5).

The *lookup impact view* displays static crosscuts that have modified one or more call sites in the base program. For instance, the developer is informed that the inter-type method declaration `Shape.setColor(SystemColor)` (Figure 5 (A)), of the aspect `aop.IntroduceToShape` has modified the call site `square.setColor(color)` in the class `base.Main` from `Shape.setColor(java.awt.Color)` to `Shape.setColor(java.awt.SystemColor)`. Similarly, the developer can observe from this



```
import java.awt.SystemColor;

public class Main {

    public static void main(String[] args)
    {
      Square square = new Square();
      SystemColor color = getSystemColor();
      square.setColor(color);

      square.draw();
      square.move(20,40);

    }
```

```
▷ aop.IntroduceToRectangle.aj
◢ aop.IntroduceToShape.aj
  ◢ Shape.setColor(java.awt.SystemColor color)
    ◢ {..} call-site:<base.Square.setColor(java.awt.SystemColor)> in <base.Main>
      ◢ before
          → base.Shape.setColor(java.awt.Color)
      ◢ after
          → base.Shape.setColor(java.awt.SystemColor)
◢ aop.ModifyHierarchySquare.aj
  ◢ declare parents: Square extends Rectangle
    ◢ {..} call-site:<base.Square.move(int, int)> in <base.Main>
      ◢ before
          → base.Shape.move(int,int)
      ◢ after
          → base.Rectangle.move(int, int)
```

```
◢ IntroduceToRectangle
  ▷ Rectangle.move(int,int)
  ◢ Rectangle.myColor
    ◢ Shadowing Impact
      ◢ [base.Rectangle].myColor
          base.Shape.myColor
          base.Rectangle.myColor
      ◢ [base.Square].myColor
          base.Shape.myColor
          base.Rectangle.myColor
```

Figure 5. The lookup and shadowing impact features of the ITDVisualiser Eclipse plugin. The *impact marker* **identifies statements in the base code whose behavior has been modified by a static cross-cutting construct.**

view that the declare parents construct `Square extends Rectangle` (Figure 5 (B)), of the aspect `aop.ModifyHierarchySquare` has modified the call site `square.move(20,40)` in `base.Main` from `base.Shape.move(int,int)` to `base.Rectangle.move(int,int)`.

The *shadowing impact view* displays constructs that have modified references to fields in the base program. For instance, the developer can observe from this view that the inter-type field declaration `Rectangle.myColor` (Figure 5 (C)), of the aspect `aop.IntroduceToRectangle` has modified all the references to the field `myColor` in classes `base.Rectangle` and `base.Square` from `base.Shape.myColor` to `base.Rectangle.myColor`.

The ITDVisualiser plug-in also supports navigation from the views to the source code, and vice versa, for reasoning about the impact of aspects. For instance, a developer can click on any node in the views to see the corresponding source code. Conversely, clicking on the *impact marker* next to an affected statement will expand the corresponding aspect in the view. The current version of the ITDVisualiser plug-in filters out static constructs that are orthogonal to the base program to avoid information overload.

5 Evaluation

In this section, we provide an experience report aimed at evaluating the effectiveness of ITDVisualiser to identify the interaction patterns between static constructs and base programs, and to enhance program comprehension in the presence of static crosscutting. To provide initial evidence of the usefulness of ITDVisualiser, we needed AspectJ systems with at least one static crosscutting construct. Our search yielded the systems in Table 1 which lists, for each system, the number of non-commented lines of code (*#SLOC*), classes (*#classes*), aspects (*#aspects*), advice (*#advice*), and inter-type declarations (*#ITDs*). The ITDs are further breaking down into inter-type field declarations (*fields*), inter-type method declarations (*methods*), and declare parents declarations (*parents*). For instance, the *ProdLine* system has 33 field declarations, 42 method declarations, and 4 declare parents declarations, for a *total* of 79 ITDs. *DCM* is a system for verifying the Law of Demeter, *ProdLine* is an implementation of the product line of graph algorithms using AOP techniques, *exptree6* is an AOP version of the expression tree program, *observer* illustrates how the Subject/Observer design pattern can be implemented with aspects, and *bean* is a system that enforces bound properties for Point objects in a Java beans context. These systems come from the Sable AspectJ benchmarks[4] and the Eclipse AspectJ examples[5]. Although small in size, these systems represent a wide array of applications of AspectJ, and have been used by other researchers for similar studies [1, 18].

For each system, we:

- Manually identified all its static crosscutting constructs,
- Manually identified the interaction patterns between each construct and the corresponding base program, and classify the construct as having either a lookup, shadowing, or orthogonal impact on the system,
- Ran ITDVisualiser on each system, and evaluated its effectiveness to correctly identify the *expected* interaction patterns for each static construct.

The results of our experiment is summarized in Table 2: the number of manually identified lookup impacts is shown in column two (#lookup-manual), the number of lookup impacts identified by ITDVisualiser is shown in column three (#lookup-tool), the number of manually identified shadowing impacts is shown in column four (#shadowing-manual), the number of shadowing impacts identified by ITDVisualiser is shown in column five (#shadowing-tool), column six (orthogonal?) states whether or not the aspects are orthogonal to the base program, and column seven (C&A) shows the minimum number of classes and aspects we had to manually examined to verify the presence or absence of impact.

In general, the ITDVisualiser tool was as effective as the manual effort in identifying the correct interaction patterns between the static constructs and the subject systems. The ITDVisualiser identified 16 lookup impacts, and 0 shadowing impacts in the ProdLine system; 27 lookup impacts, and 0 shadowing impacts in the DCM system; 4 lookup impacts, and 0 shadowing impacts in the exptree6 system; and no lookup or shadowing impact in both the bean and observer systems. To manually verify the interaction patterns and impacts in ProdLine, we had to examine 20 of its 21 classes and aspects since inter-type declarations are heavily used, 79 ITDs in total. Similarly, 28 of the 34 classes and aspects of the DCM system had to be examined to verify impact. The difficulty to manually reason about the impact of such large number of ITDs on base programs further underscores the need for tools such as ITDVisualiser.

Both the ITDVisualiser and our manual investigation revealed no shadowing impacts in the systems we studied in spite of the heavy use of inter-type field declarations, 37 in total. This is a good thing since shadowing impacts are generally an indication of bugs. A report such as this will either give developers confidence that their use of field declarations have not shadowed existing fields in a system, or assist them to immediately identify and fix cases of field shadowing.

The lookup impacts in all the subject systems were caused by cases of method overriding. We encountered no cases of method overriding based on subtypes. The reason for this may be that lookup impacts caused by method overriding based on subtypes usually leads to bugs, and since these benchmarks are well tested, such problematic impacts may have been identified and eliminated. Notwithstanding, the ITDVisualiser tool was able to identify all the non-buggy impacts, and is therefore reasonable to assume that it will equally identify problematic shadowing and lookup impacts when used in future AOP development.

[4]http://www.sable.mcgill.ca/benchmarks/

[5]http://www.eclipse.org/aspectj/doc/released/progguide/examples.html

Table 1. Description of the subject systems

					#ITDs			
Systems	SLOC	#classes	#aspects	#advice	fields	methods	parents	total
ProdLine	1345	9	11	15	33	42	4	79
DCM	3435	30	4	8	0	1	1	2
exptree6	327	7	5	4	1	25	1	27
bean	276	2	1	2	1	5	1	7
observer	268	6	2	1	2	7	2	11

Table 2. A summary of the types of impact of aspects on the subject systems

Systems	#lookup-manual	#lookup-tool	#shadowing-manual	#shadowing-tool	orthogonal?	C&A
ProdLine	16	16	0	0	NO	20
DCM	27	27	0	0	NO	28
exptree6	4	4	0	0	NO	11
bean	0	0	0	0	YES	2
observer	0	0	0	0	YES	6

6 Related Work

This paper builds on previous works related to the interaction patterns between aspects and base programs, and change impact analysis for AspectJ. We review these related works in this section of the paper.

Interaction patterns between aspects and base programs: Rinard et al. [13] identified four interaction patterns between *advice* (i.e., the dynamic constructs of AspectJ) and base programs, and developed analysis techniques capable of automatically recognizing these patterns. They classified an advice as either being an *augmentation* (in which the entire code of the method affected by an advice always executes), a *narrowing* (in which the entire code of the method affected by an advice either executes, or none of the code is executed), a *replacement* (in which the entire code of the method affected by an advice is never executed), or a *combination* of the preceding three. Similarly, Clifton et al. [4] classified advice as either being a *observers* or an *assistant*. Observers have purely observational interactions with the methods they affect, whereas assistants can change the specification of the methods they interact with. The similarity between our work and that of both Renard et al. and Clifton et al. is that we all provide a set of criteria for classifying the interaction patterns between aspects and base program. However, theirs is exclusively focused on dynamic constructs, whereas our work deals with static constructs, and is therefore complementary to theirs.

The AspectJ Development Tools (AJDT) [6] of Eclipse provide extensive visualization support for understanding the impact of advice on base program; however, very limited support is provided for the static crosscutting constructs. For instance, given the construct `declare parents: classX extends classY`, the *Cross References* view and side bar annotations of AJDT communicates the obvious — that `classX` now extends the `classY`. AJDT does not communicate the actual impact of such hierarchy modification on statements in the base program. *ITDVisualiser* complements AJDT by identifying the statements in a base application affected by each inter-type declaration, and by providing an explanation for the potential change in the behavior of the application.

Change Impact analysis in AspectJ: Zhang et al. [18] studied change impact analysis for AspectJ systems. Their approach is based on *atomic changes* and *aspect-aware* call graphs. Atomic changes capture the semantic differences between two versions v_i and v_{i+1} of an AspectJ program — that is, all the edit operations required to obtain version v_{i+1} of a program from version v_i. Their aspect-aware call graph algorithm is similar to ours — the call graph is constructed from the woven code — with one difference: advice are treated as methods and included in the call graph. Zhang et al. developed static analysis techniques which use the atomic changes and the aspect-aware call graph to identify program fragments and test cases affected by a modification tasks. Storzer et al. [14] also proposed the use of call graphs to identify test cases affected by static crosscuts given two different versions an application, but have yet to provide an implementation their ideas. Zhao [19] proposed an approach which used static slicing and an aspect-oriented dependence graph to identify statements affected by changes to an AspectJ program, but have yet to implement such a static slicer for AspectJ. The works of Zhang et al. and Storzer et al. are concerned with regression test selection, and is therefore coarse grain with little explanation of the changes in the behavior of the base program. Our approach is concerned with the interaction patterns between static constructs and base programs, is fine grain (all statements affected by static crosscutting constructs are identified), and also provide possible reasons for the changes in the behavior and state of statements in the base program.

7 Conclusions

Simple structural modifications such as adding a new method or field from an aspect to an object-oriented application might adversely distort the behavior or state of a system. Inherent properties of aspect and object-oriented languages such as obliviousness and subtype polymorphism makes reasoning about the behavior of these systems difficult and error prone. There is therefore the need for tools capable of identifying points of interaction between aspects and base applications, and explaining the corresponding interaction patterns in order to facilitate program comprehension. This paper addressed this challenge for the static crosscutting constructs of AspectJ — an aspect-oriented language for Java. We have identified patterns of interactions between static constructs and base applications, and have developed a static analysis and visualization tool called ITDVisualiser to automatically identify these patterns. Our analyses not only identifies statements in a base program whose behavior have been modified by static constructs, but also provide possible explanations for the change in behavior. Our experience with ITDVisualiser indicates that it performs as well as manual classification in identifying the interaction patterns between aspects and base programs. Tools such as ITDVisualiser are necessary not just for program comprehension, but also as a safety net against unintended aspect-induced structural modifications.

Acknowledgments

The authors would like to thank Eric Bodden for his support with the use of abc and the Soot Framework. This work was supported by NSERC.

References

[1] P. Anbalagan and T. Xie. Automated inference of pointcuts in aspect-oriented refactoring. In *Proceedings of the 29th International Conference on Software Engineering*, pages 127–136, 2007.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 87–98, 2005.

[3] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with ajdt. In *Workshop on the Analysis of Aspect-Oriented Software*, 2003.

[4] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *In the Foundations of Aspect Languages*, pages 33–44, 2002.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communication of the ACM*, 44(10):29–32, 2001.

[6] T. E. Foundation. Aspectj development tools subproject. www.eclipse.org/ajdt/.

[7] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Communication of the ACM*, 44(10):59–65, 2001.

[9] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, pages 49–58, 2005.

[10] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[11] G. A. D. Lucca, M. Smit, B. Fraser, E. Stroulia, and H. J. Hoover. Comprehending aspect-oriented programs: Challenges and open issues. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 286–292, 2007.

[12] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 90–99, 2003.

[13] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, pages 147–158, 2004.

[14] M. Storzer and J. Krinke. Interference analysis for AspectJ. In *Proceedings of Workshop FOAL 2003, held in conjunction with AOSD 2003*, 2003.

[15] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 264–280, 2000.

[16] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[17] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Celadon: A change impact analysis tool for aspect-oriented programs. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Engineering*, 2007.

[18] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for aspectj programs. *International Conference on Software Maintenance*, pages 87–96, 2008.

[19] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 108–112, 2002.