A CORE GENERIC META-MODEL FOR ASPECT-ORIENTED PROGRAMMING LANGUAGES

FARHANA EVA ALAM

A CORE GENERIC META-MODEL FOR ASPECT-ORIENTED PROGRAMMING LANGUAGES

by

© **Farhana Eva Alam**

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

September 2010

St. John's, Newfoundland, Canada

Abstract

Aspect Oriented Software Development (AOSD) has its roots in the need to deal with requirements that cut across the primary modularization of a software system. On the programming level, mature, industrial-strength tools like the de-facto standard AspectJ exist. However, on the modeling level, there is as yet little support for AOSD. Previous work, which was platform specific, has provided support for only AspectJ. However, as AspectJ does not support dynamic aspect-orientation, the developed model only provides support for static AOSD. Building on previous work, using standard UML extension mechanisms, this research develops UML modeling support for both static and dynamic AOSD. Comparing language and aspect-oriented features of AspectJ, AspectS and AspectML, as a first step to our generic profile, we present a profile which supports only static part of AspectJ and AspectS. This helps us to figure out the modeling elements that are required for dynamic profile but missing in the current profile. As the second step, a generic but only dynamic profile (does not provide support for static AOSD) is presented. These two profiles clearly show the difference between static and dynamic AOP in modeling level. We use the above steps and develop the final generic profile that allows existing UML tools to express AOSD models. The developed model ensures modeling support for both static and dynamic AOSD from the same profile. To verify the necessity and correctness of the profiles used as working steps, we apply each of those to several examples. Furthermore, the generic profile is applied to examples from AspectJ and AspectS to make sure that it can express both static and dynamic AOSD. Code generation is done by working from the UML XMI (XML Model Interchange) format, the standard UML serialization.

This is one of the standardized mechanisms and is therefore compatible with existing modeling tools. Existing work has demonstrated the use of XSLT (XML Stylesheet Language Transforms) for generating XMI to AspectJ code. We leverage that mechanism. As a proof-of concept, we implement XSLTs that generates valid code for our target languages (AspectJ, AspectS).

Acknowledgments

I would like to express my deepest gratitude to my supervisors, Dr. Joerg Evermann and Dr. Adrian Fiech. It has been my good fortune to have the opportunity of working with them. I would like to thank them their remarkable help, encouragement and support to carry on my research work.

I am also deeply grateful for the financial support of the School of Graduate Studies, Memorial University. Further, a special thanks to the Department of Computer Science, without its resources and administrative help it would not be possible to pursue my Master's degree at Memorial University. In addition, I am also grateful to the writing center of Memorial University for checking the entire thesis.

I want to express my endless gratitude towards my parents, sister, in-law family members and relatives for inspiring me with their unbound love, faith, and support to continue my graduate school career. My sincere appreciation to my husband Refaul Ferdous who gave me all kinds of support and taken extra care of me and our five months old baby girl Waniya who was born at the middle of my thesis writing.

And finally, looking back over my study period at Memorial University, this thesis would not be possible without the support of few peoples. Specially, I would like to thank Morsheda Mamataz, Kamrunnahar Eamy, Nigar Sultana, Madhabi Roy, Wendy Khandakar, Naushaba Sheikh, Asma Dewan, Negar Noor, Shazli Khan, Kazi Tayubul Haq, Reedwanul Islam, Wasimul Bari, Shakil Ahmed, Shibly Rahaman, Matthias Tilsner, and Riobard Zhan for their moral support and help. They also have made my life here more joyful and happy.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

INTRODUCTION

1.1 Separation of Cross-cutting Concerns

Separation of concerns (SoC) is a fundamental principle in software engineering. It entails breaking down a program into distinct parts called concerns. Concerns are some specific requirements that must be addressed to satisfy the overall system goal while designing a system. A set of concerns that compose the entire software system can be classified into two categories: core concerns and cross-cutting concerns [1]. Core concerns capture the basic functionality of a system, whereas the cross-cutting concerns capture the features shared by many of the core concerns. All programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing procedures, modules, objects, classes or methods. For example, languages like C++, Java and C#, which belong to the family of object-oriented programming (OOP) languages, support modularizing of core concerns. These languages can separate the concerns by encapsulating them into objects. However, features like logging, persistence, and security disobey this form of encapsulation as they are shared among many of the core concerns and cannot easily be fitted into the OOP approach [1, 2, 3]. Furthermore, the OOP paradigm often forces the designer to create a coupling among the core concerns and the cross-cutting concerns in the software system. A program implemented in Java for a Shopping Center which includes items, inventory, and carts can be a simple example that clearly shows how the crosscutting concerns like logging are not modularized in OOP implementations. We assume that the program consists of several classes like Item, Inventory, and ShoppingCart, each of which contains multiple methods. To see which methods are being executed during runtime, a programmer needs to implement some logging feature such as print statement within each method in this program as shown in Figure 1.1.

Additional codes in each Class

```
import java.util.*;

public class Inventory {

    private List<Item> _items = new Vector<Item>();

    public void addItem(Item item) {
        Date date=new Date();
        System.out.println(date+" Inventory addItem"+"\n INFO: Entering");
        _items.add(item);
    }

    public void removeItem(Item item) {
        Date date=new Date();
        System.out.println(date+" Inventory removeItem"+"\n INFO: Entering");
        _items.remove(item);
    }

}
```

Figure 1.1: Additional codes in each method of Inventory class

The same printing statement, which can be considered as cross-cutting concern, is added to all the methods. These printing statements cannot easily be modularized as a separate entity and does not remain in the form of encapsulation. Section 1.2 illustrates the consequences of non-modularized cross-cutting concerns in software design and implementation.

1.2 Modularization of Cross-cutting Concerns

Cross-cutting concerns that are non-modularized often lead to code tangling or code scattering. Code tangling is caused when multiple concerns are considered several times while implementing a module. On the other hand, code scattering occurs while implementing a single concern in several modules. The example of the shopping-cart program (Section 1.1) shows code scattering, since the same printing statement logging feature is implemented in several classes. However, implementing features like security and persistence in each class of the same example will introduce code tangling in the same system.

A program containing either code scattering or code tangling is always difficult to be traced or evaluated, since adding some new code or doing some minor modifications may require the programmer to edit the entire program. It also reduces the chances of code reuse. Furthermore, developing a system with these non-modularized concerns is always time consuming and may contribute to low productivity in software development [1]. For example, if someone wants to change the logging messages in the shopping-cart example then he needs to modify all the existing print statements. Thus code tangling and code scattering cause undesirable system complexity and should be avoided.

1.3 Aspect-oriented Programming

There are many programming problems involving important concerns like transactions, security, distribution, or logging where OOP techniques are not sufficient to clearly capture all of the important design decisions the program must implement [2, 1]. The need for modularizing and encapsulating those concerns gives rise to the concept of aspect-oriented programming (AOP). AOP provides a new programming technique that clearly expresses programs involving cross-cutting concerns, including appropriate isolation, composition and reuse of the code [2].

1.3.1 AOP Methodology

The methodology for developing a system in AOP is similar to other methodologies. A system developed with AOP requires the following steps [1]:

a) Identifying the concerns:

In this step, the requirements are decomposed to identify core concerns and cross-cutting concerns of the system. For example, the core concerns of the shopping-cart example are the concerns related to item, shopping cart, inventory and shopping cart operator. The crosscutting concern for this example is the logging functionality that is scattered and repeated in the core modules.

b) Implementing the concerns:

In this step, each concern identified in the previous step will be implemented separately. For the core concerns, the base language of that particular AOP language is chosen for implementation. On the other hand, cross-cutting concerns are implemented separately in stand-alone units called *aspects* (see Section 1.3.2). For example, if Java is chosen as the base methodology for the shopping-cart problem then the core concerns are implemented as classes excluding the logging functionality. However, being a cross-cutting concern, logging functionality will be implemented as a separate unit, known as an *aspect.*

c) Forming the final system:

In this step, the final system is formed according to some rules known as "weaving rules" specified within the aspect. An aspect *weaver*, which is a compiler-like entity, composes the final system by combining the core and crosscutting modules through a process called *weaving.*

Figure 1.2, taken from [1], shows the difference between two different implementations of a banking system with logging functionality. In both implementations, the system includes several client modules such as the accounting module, the ATM module and the database module. The first implementation is done using OOP methodology, where the code tangling occurs between all the modules needing the logging module [1].

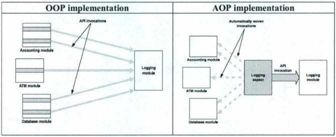


Figure 1.2: Implementation of a banking system (Redrawn after [1])

AOP implementation for the same banking system (right part of Figure 1.2) shows the modularization of logging functionality where none of the clients contain code for logging. Furthermore, logging becomes modularized as the crosscutting logging requirements are now mapped directly only to the logging aspect. With such modularization, any changes to requirements affect only the logging aspect but not the clients. Using AOP methodology, client code remains completely isolated and unchanged. Thus, AOP facilitates programs with improved traceability, higher modularization, easier system evolution and more code reuse.

1.3.2 AOP Language Implementation

As the AOP paradigm is gaining much industry support, we have several AOP implementations for existing programming languages. For example, AspectJ [3] for Java, AspectC++ for C++ [4], Aspect# for C# [5], AspectS [6] for SmallTalk/Squeak [7] and AspectML [8] for ML are some of the popular AOP language implementations.

1.3.2.1 Basic Terminology:

The followings are some standard terms used in Aspect-oriented programming:

a) Join point:

A *join point* is a point in the dynamic control flow of an application. Thus a join point can for instance represent a call to a method, execution of a method, the event of setting a field or the event of handling an exception. For example, AspectJ exposes several categories of join points such as: method join points, constructor join points, field access join points and class initialization join points [1]. On the other hand, it does not expose join points for loops, super calls, throws clauses, multiple statements and the like.

b) Pointcut:

A *pointcut* is a means to specify the weaving rules. It selects join points that satisfy those rules and collects program execution context at those points. Some AOP language implementations such as AspectJ [3] have a special language construct for pointcuts, whereas some other languages like AspectS [9] do not have any dedicated pointcut language.

c) Advice:

Advice is a means of specifying code to run at a join point that has been selected by a pointcut.

d) Aspect:

The combination of one or more point-cuts and advices is termed an *aspect*.

e) Static crosscutting:

In AOP, we often find that in addition to affecting dynamic behavior using advice, it is necessary for aspects to affect the static structure in a crosscutting manner. While dynamic crosscutting modifies the execution behavior of the program, static crosscutting modifies the static structure of the types—the classes, interfaces, and other aspects—and their compile-time behavior.

1.3.2.2 Join Point Model

Each AOP implementation has its own Join Point Model (JPM) which defines three things:

(i) When the advice can run,

(ii) A way to specify pointcuts and

(iii) A means of specifying code to run at a join point.

JPMs of different AOP implementations can be compared based on the following criteria:

- The join points exposed,
- How pointcuts are specified,

- The operations permitted at the join points, and
- The structural enhancements that can be expressed such as static crosscutting.

The AOP implementations of some of the languages are described in detail in the next chapter.

1.4 Aspect-oriented Modeling

Aspect-oriented modeling (AOM) supports modularization of cross-cutting concerns at the software design level. Most AOM techniques focus on providing modeling capabilities for the core AOSD concepts, usually as extensions to the Unified Modeling Language (UML).

UML [10] is a standardized general-purpose modeling language in the fields of software engineering. UML is managed, and was created by the Object Management Group (OMG). To date UML is not only used to model application structure, behavior, and architecture, but also business process and data structure.

UML, along with the Meta Object Facility (MOF)[1], provides a key foundation for OMG's Model-Driven Architecture, which unifies every step of development and integration from business modeling, through architectural and application modeling, to development, deployment, maintenance, and evolution [10].

A model can be either platform-independent or platform-specific. A platform-independent model (PIM) is a model of a software system or business system that is independent of the specific technological platform used to implement it. A platform-

[1] The Meta-Object Facility (MOF) is an Object Management Group (OMG) standard for model-driven engineering.

specific model (PSM) is a model of a software or business system that is linked to a specific technological platform (e.g. a specific programming language, operating system or database).

UML allows both PIM and PSM. Besides this, it provides a mechanism known as profile. Profiles are the standard mechanism to extend UML. The profile mechanism exists within UML so models applying a profile are fully UML compatible. A UML model applying a profile is UML, and any UML tool can process it [11].

While there has been prior work on extending UML to AOM, most of the extensions expand UML either by introducing new meta-model classes or new notation elements without providing meta-level support. Furthermore, many of the existing AOM approaches are programming language specific and allow modeling on the platform specific model (PSM) level. While AOP language implementations are rapidly maturing, a platform independent model is necessary to increase the reusability of system. It will also ease the communication between developers from different backgrounds. Building on previous work [12], this research presents a core generic meta-model, which is a profile based on the core features of some AOP languages. Core features are selected by comparing some of the AOP language extensions.

As AOP paradigm is rapidly maturing we have AOP language extensions for many of the existing programming languages such as Java, C, C++, SmallTalk, Haskell, ML, PHP and XML. However, in this research we only study the features of AspectJ, AspectS and AspectML. AspectJ is chosen because it is considered as the first complete and powerful language extension for AOP. It possesses a wide variety of AOP language features. Features of other AOP language extensions (e.g. AspectC,

AspectC++) for existing languages that follow OOP approach (E.g. C, C++) can be considered as subsets of the features of AspectJ.

Like AspectJ, AspectS follows OOP approach. However, because AspectS supports dynamic AOP (will be discussed in Chapter 3), it is also considered for this study. This is a fact that the use of several languages provides some generality. As a result, we wanted to keep a AOP language extension of an existing language (e.g. Haskell, ML, etc) that belongs to the family of functional language. Since AspectML is well developed and widely used among these languages, it is also included in our study.

1.5 Objectives

The objectives of this research are:

- To study AOP languages in order to identify their similarities and differences.
- To develop a platform independent UML-based model (PIM), which will be a UML profile for the AOP paradigm. The model will allow aspect modeling to be used within existing, mature software tools.
- To develop an example of a transformation from a PIM to platform-specific code.
- To find a way to handle the features unique to the AOP languages that we are studying.

1.6 Thesis Structure

Chapter 1 introduces AOP paradigm. It shows the necessity of the paradigm. It also introduces both aspect-oriented programming and aspect-oriented modeling along with the basic terminology used for these two.

Chapter 2 provides a broad overview of AOP languages that we study. This helps the reader who is not very familiar with AOP to develop some idea of AOP languages. A running example and its implementation in three different languages allow us to have a comparative picture of the languages that we study.

Chapter 3 focuses on two different AOP approaches: static and dynamic. Since the languages that we study do not follow a single approach from these two, this chapter provides a detailed overview of these approaches. It also does explain the difference between dynamic and static AOP.

Chapter 4 compares the three languages that we study. This chapter provides a detailed overview of different features of each language. The running example is used to demonstrate how the different languages can be used to implement a solution to the same problem. Thus it helps to select the core features of AOP languages.

Chapter 5 elaborates the discussion on AOM. It includes some of the prior works related to our research. It also aims to develop a generic UML "profile" that can be used to model aspect-oriented system.

Chapter 6 demonstrates the application of the developed profile. It illustrates base model for five examples. The developed profile is applied to both static and dynamic model for each example.

Chapter 7 presents code generation for each application shown in Chapter 6. It presents XSLTs to generate AspectJ and AspectS code from the model to which the generic profile is applied.

Chapter 8 focuses on overall discussion on this research. It includes the importance and limitations of the developed profile.

Chapter 9 presents the summary and conclusion of this research work. It also highlights some possible future works that can be done by extending this research.

Chapter 2

AN OVERVIEW OF AOP LANGUAGES

2.1 Running Example

The Shopping-Cart problem given in [1] is an example that clearly shows how code scattering occurs in a conventional implementation when some features such as logging are added to it. It also reveals the necessity of modularizing cross-cutting concerns. The basic functionalities of Shopping-Cart without logging are implemented using the following classes: Item class, Inventory class, ShoppingCart class, ShoppingCartOperator class and a Test class.

Listing 2.1 represents the *Item* Class. This class models the items which can be purchased. This class has a constructor and three public methods: *getId()*, *getPrice()* and *toString()*. The *getId()* and the *getPrice()* methods provide the identifier and price of the item respectively. The *toString()* method simply sets the format of an item to a string.

Listing 2.1 *Item* class

```
public class Item {
  private String _id;
  private float _price;
  public Item(String id, float price)
 {
   _id = id;
   _price = price;
  }
  public String getID() {
   return _id;
  }
  public float getPrice() {
   return _price;
  }
  public String toString() {
   return "Item: " + _id;
  }
 }
```

The *Inventory* class, shown in Listing 2.2, represents the list of items available for purchasing. This class has two public methods: *addItem()* and *removeItem()*. Both of these methods take an item as argument which can be added to or removed from the existing item inventory using these two methods.

Listing 2.2 *Inventory* class

```
import java.util.*;
 public class Inventory {
 private List _items = new Vector();
 public void addItem(Item item) {
  _items.add(item);
 }
 public void removeItem(Item item) {
  _items.remove(item);
 }
}
```

The *ShoppingCart* class shown in Listing 2.3 represents the list of items in a shopping cart of a customer. The two public methods *addItem()* and *removeItem()* are respectively used to add and delete specific items from the shopping cart's item list.

Listing 2.3 *ShoppingCart* class

```
import java.util.*;
public class ShoppingCart {
private List _items = new Vector();
public void addItem(Item item) {
  _items.add(item);
 }
public void removeItem(Item item) {
  _items.remove(item);
 }
}
```

The *ShoppingCartOperator* class is used to model the operations related to a purchase. This class has two static public methods: *addShoppingCartItem()* and *removeShoppingCartItem()*. The purpose of these two methods is to update both lists: inventory and shopping cart. The method *addShoppingCartItem()* is used to model a "purchase of an item" by adding an item to the shopping cart and deleting the same item from the inventory. On the other hand *removeShoppingCartItem()* is used to model a "return of an item" by removing an item from the shopping cart and adding it back to the inventory. The entire *ShoppingCartOperator* class with all the above mentioned methods is shown below in Listing 2.4.

Listing 2.4 ***ShoppingCartOperator*** **class**

```
public class ShoppingCartOperator {
  public static void addShoppingCartItem(ShoppingCart sc,
                                   Inventory inventory, Item item){
    inventory.removeItem(item);
    sc.addItem(item);
  }
  public static void removeShoppingCartItem(ShoppingCart sc,
                                   Inventory inventory,Item item) {
    sc.removeItem(item);
    inventory.addItem(item);
  }
}
```

Listing 2.5 shows the *Test* class that tests the functionality of the classes discussed above. This class does not print any text message as output. However, within the class, at first, three items are added to the inventory list. Then, using the method addShoppingCartItem of *ShoppingCartOperator* class, two of the previously added item are added to the list of shopping cart, and are deleted from the list of inventory.

Listing 2.5 ***Test*** **class**

```
public class Test {
  public static void main(String[] args) {
  Inventory inventory = new Inventory();
  Item item1 = new Item("1", 30);
  Item item2 = new Item("2", 31);
  Item item3 = new Item("3", 32);
  inventory.addItem(item1);
  inventory.addItem(item2);
  inventory.addItem(item3);
  ShoppingCart sc = new ShoppingCart();
  ShoppingCartOperator.addShoppingCartItem(sc, inventory, item1);
  ShoppingCartOperator.addShoppingCartItem(sc, inventory, item2);
   }
}
```

The *Test* class can be modified to trace some of the method execution. This can be done in a similar fashion as in [1] by using the library class provided in Java for logging. Listing 2.6 shows the modified version of *Test* class with logging functionality.

Listing 2.6 *Test* class (modified version-2)

```
import java.util.Date;
public class Test {
public static void main(String[] args) {
   Date date=new Date();
   System.out.println(date+" Test main"+"\n INFO:
Entering");
     //rest of the method's body should be same
  }
}
```

Instead of using the Java library class *Logger*, for simplicity, better performance, and code reuse (in later Sections), we use some conventional print statements in the above example. In order to see the execution information as shown in Figure 2.1, it is required to write the same type of logging code inside each related class and its methods.

```
Wed Apr 15 21:24:49 NDT 2009 Test main
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 Inventory addItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 Inventory addItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 Inventory addItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 ShoppingCartOperator addShoppingCartItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 Inventory removeItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 ShoppingCart addItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 ShoppingCartOperator addShoppingCartItem
 INFO: Entering
Wed Apr 15 21:24:49 NDT 2009 Inventory removeItem
 INFO: Entering
Wed Apr 15 21:24:50 NDT 2009 ShoppingCart addItem
 INFO: Entering
```

Figure 2.1: Output of the shopping-cart program with "logging"

The repetition of this same logging code in several classes of this system is introducing code scattering. Further modification to this system can also lead to more forms of code scattering and code tangling. AOP language implementations provide the techniques to solve the code tangling and code scattering problems. Brief overviews of some of the AOP language implementations are given in the following sections along with some examples. Our running example shopping-cart is also implemented in each of the discussed AOP languages.

2.2 AspectJ

2.2.1 An Overview of AspectJ

AspectJ [3] is a widely used, general purpose, aspect-oriented language extension to the Java programming language. Being an extension to Java, every valid Java program can be executed as a valid AspectJ program. AspectJ adds to Java new constructs to specify the weaving rules programmatically: *aspect, join point, pointcut, advice, introduction and compile-time declaration.* The class files produced by an AspectJ compiler conform to the Java byte-code specification. As a result, these class files can be executed by any compliant Java virtual machine (VM). Since it uses Java as the base language and retains all the benefits of Java, it is easy for programmers from a Java background to understand the AspectJ language. An AspectJ file should be saved with .aj extension and can include constructs from Java and AspectJ in it.

2.2.1.1 Aspect

In AspectJ, an aspect is like a class in Java. It can include data members and methods and can have access specifications, but it cannot be instantiated directly. An aspect can have an access specifier (visibility) of "privileged" in order to read and write the private members of the classes it is crosscutting [1]. It can extend classes and abstract aspects, as well as implement interfaces. However, to reduce complexity, aspect inheritance is limited to only inheriting from abstract aspects but not from concrete aspects [3]. Moreover, an aspect can be embedded inside classes and interfaces as a nested aspect. Figure 2.2 shows an AspectJ file where aspect HelloWorld is declared as a public aspect. It contains a pointcut and an advice in its body. However it is possible to modify the aspect by including some regular variables and methods in it.

*HelloWorld.aj

```
public aspect HelloWorld {

    pointcut deliverMessage()
        : call (* Test.deliver(..));
    before(): deliverMessage(){
        System.out.print("Hello World! ");
        }
}
```

Figure 2.2: An AspectJ file named HelloWorld

2.2.1.2 Join Point

AspectJ allows adding new behavior in some special parts or areas known as join points of a program. In AspectJ, the join points are mainly exposed as: method join points, constructor join points, field access join points, exception handler execution join points, class initialization join points, object initialization join points, object pre-initialization join points and advice execution join points. On the other hand, loops, super calls, throws clauses, or multiple statements are not exposed as join points in AspectJ. Each of the above mentioned categories of exposed join points can include one or more types of join points. For example, the method join point category exposes both method calls and executions. Figure 2.3 shows a Java class named *Test* in which both the method call and method execution join points are exposed when an AspectJ file is used in parallel to select those as pointcuts.

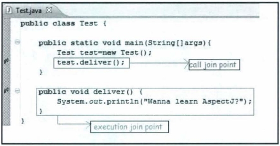


Figure 2.3: A class named Test in Java

2.2.1.3 Pointcut

In AspectJ, a pointcut can both specify a single join point in a system and match a set of join points. It can be either *anonymous* or *named*. A named pointcut is defined using the keyword *pointcut* and can have access specifiers. On the other hand, an anonymous pointcut can be specified as part of an advice [1].

Join points in AspectJ are specified using pointcut designators. The pointcut designators match join points in AspectJ either by capturing join points based on the category to which they belong or by capturing join points based on matching the execution circumstances under which they occur. AspectJ allows the following types of pointcuts:

a) Kinded pointcuts:

Each exposed join point mentioned earlier has a specific pointcut designator, which is used to capture a join point from the program flow. However, all of these pointcut designators follow a particular format of syntax. As a result, they fall into the category of kinded pointcut. Table 2.1, taken from [1] shows the mapping between each exposed join point and its corresponding pointcut designator.

Table 2.1: Mapping of exposed join points to pointcut designators

Join Point Category	Pointcut Syntax
Method execution	`execution(MethodSignature)`
Method call	`call(MethodSignature)`
Constructor execution	`execution(ConstructorSignature)`
Constructor call	`call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read access	`get(FieldSignature)`
Field write access	`set(FieldSignature)`
Exception handler execution	`handler(TypeSignature)`
Object initialization	`initialization(ConstructorSignature)`
Object pre-initialization	`preinitialization(ConstructorSignature)`
Advice execution	`adviceexecution()`

The signature to be used in each pointcut designator is clearly described in both [1] and [13] along with examples. However, the following (Figure 2.4) is a simple example of a named pointcut where the name of the pointcut is `deliverMessage`. As the signature part contains a method signature, looking at the above table it is clearly visible that this pointcut selects a method call join point, i.e. the call to the *deliver* method of *Test* class showed in Figure 2.3.

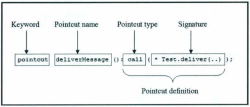


Figure 2.4: An example of a named pointcut (adapted from [1]

b) Control-flow based pointcuts

These pointcuts capture join points based on the control flow of join points captured by another pointcut [1]. In AspectJ, control-flow based pointcuts can be categorized as follows:

i. cflow()

The *cflow()* pointcut takes a pointcut as argument and captures all the join points in the control flow of the specified pointcut, including the join points matching the pointcut itself.

ii. cFlowBelow():

The *cFlowBelow()* pointcut takes a pointcut as argument and captures all the join points in the control flow of the specified pointcut, excluding the join points matching the pointcut itself.

c) Lexical-structure based pointcut

Lexical-structure based pointcuts capture join points occurring inside a segment of source code of specified classes, aspects and methods [1]. There are two pointcuts in this category:

i. within()

The *within()* pointcut is used to capture all the join point specified within the body of the specified classes or aspects, including the nested classes[1].

ii. withincode()

The *withincode()* pointcuts can have the forms like withincode(MethodSignature) or withincode(ConstructorSignature). They are used to capture all the join points inside a lexical structure of a method or a constructor, as well as any local classes in them [1].

d) Execution object pointcuts

These pointcuts match the join points based on the types of the objects at execution time. They are also used to expose the context of the specified join point. This category consists of the following pointcuts:

i. this()

this refers to the current object. Therefore, the *this()* pointcut selects all the join points associated with the current object.

ii. target()

target refers to the object on which the method is called. The *target()* pointcut is usually used with method call join point. It is used to select the join points associated with the object on which the method is invoked.

e) Argument pointcuts (args())

The *args()* pointcut can expose the context at the matched join point. The empty bracket after the name of the pointcut indicates that it does not expose the context of the join point.

f) Conditional check pointcuts (if())

Conditional check pointcut captures join points on the basis of some conditional check at the join point [1].

2.2.1.4 Advice

Advices of AspectJ are similar to methods of Java. But they have some differences as well. Such as:

- Does not have a name
- Cannot be called directly (it is the system's job to execute it)
- Does not have an access specifier

AspectJ allows different types of advices as a means to specify code to run at a join point.

Advice is invoked automatically by the AOP runtime when the pointcut matches the join point. AspectJ supports following three types of advices:

a) Before advice:

Before advice allows adding new behavior before the specified join point.

b) After advice:

After advice allows adding new behavior after the specified join point.

c) Around advice:

Around advice has the ability to bypass the join point, allow it to execute as it is, or execute it with changed context.

2.2.1.5 Static Crosscutting

AspectJ allows static crosscutting (also known as *introduction). Introduction* affects the static structure of programs through crosscutting. Using *introduction* it is possible to introduce changes to the classes, interfaces, and aspects of the system [1].

2.2.2 Running Example in AspectJ

When implementing the running example ShoppingCart in AspectJ, the base model is kept without any logging code. As shown in Listing 2.7, the crosscutting concerns are modularized in an aspect named *TraceAspect*. The pointcut named traceMethods() selects all the method execution join points specified by the wildcards in the program flow, except those within the lexical scope of the *TraceAspect* aspect.

Listing 2.7 ***TraceAspect*** **Aspect in AspectJ**

```
import java.util.Date;
import org.aspectj.lang.*;
 public aspect TraceAspect {
 pointcut traceMethods()
 : execution(* *.*(..)) && !within(TraceAspect);

 before() : traceMethods() {
 Date date=new Date();
 Signature sig = thisJoinPointStaticPart.getSignature();
 System.out.println(date+" "
+sig.getDeclaringType().getName()+ " "+sig.getName()+"\n
```

```
INFO: Entering");
    }
}
```

The pointcut *traceMethods()* is then advised by the before advice, which places the logging code before each of the method execution specified by the pointcut itself. Figure 2.5 shows the output of the ShoppingCart example implemented in AspectJ. Although this implementation produces the same output as the one (Figure 2.4) produced by the example implemented without aspects, the difference here is that a programmer does not need to write logging code in each method of each class. As a result, in the current implementation the crosscutting concerns are modularized. Therefore, the logging feature does not create code tangling or scattering in this example.

```
Wed Jun 03 00:53:23 NDT 2009 Test main
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 Inventory addItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 Inventory addItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 Inventory addItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 ShoppingCartOperator addShoppingCartItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 Inventory removeItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 ShoppingCart addItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 ShoppingCartOperator addShoppingCartItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 Inventory removeItem
 INFO: Entering
Wed Jun 03 00:53:24 NDT 2009 ShoppingCart addItem
 INFO: Entering
```

Figure 2.5: Output of the ShoppingCart example in AspectJ

2.3 AspectS

2.3.1 An Overview of AspectS

AspectS [6] is an AOP implementation for the Smalltalk or Squeak[2] environment. It mainly draws on the results of two projects: AspectJ[3], and John Brant's MethodWrappers[3] [14]. In its current implementation, AspectS is realized using plain Smalltalk only, changing neither Smalltalk's syntax nor its virtual machine. As a result, AspectS also complies with following language properties of SmallTalk:

- Everything (e.g. class, instances of class, etc) is an object.
- All communications are done using method passing between objects.
- Classes inherit via single inheritance.
- The default access specification is as follows:

Table 2.2: Access Specification

AspectS	Acts like	Java
Variables	←→	Private variables
Instance method	←→	Public methods
Class methods	←→	Public Static

All the examples related to AspectS are written in this text using Squeak version 3.8 and AspectS version 0.6.6. AspectS version 0.6.6 is available in [9].

[2] Squeak is an open, highly portable Smalltalk-80 implementation. Its virtual machine is written entirely in Smalltalk. The terms Squeak and Smalltalk are used interchangeably in this text.

[3] MethodWrappers is a powerful mechanism to add behavior to a compiled Smalltalk method

The packages, classes and methods in Squeak can be found in Squeak *system browser* shown in Figure 2.6.

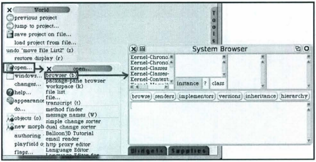


Figure 2.6: Opening system browser in Squeak

We create a new category named *HelloWorld*. Then we create a class named *Test* inside the category *HelloWorld* as shown in Listing 2.8.

Listing 2.8 *Test* class in Squeak

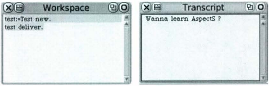
```
Object subclass: #Test
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'HelloWorld'
```

A method named *deliver* is created as shown in Listing 2.9.

Listing 2.9 *deliver* method of *Test* class in Squeak

```
deliver
  Transcript show:' Wanna learn AspectS ? '.
```

The code shown in the workspace[4] (Figure 2.7 a) if executed, produces output in the transcript[5] window (Figure 2.7 b).



a. Test code for Hello World   b. Out put of the Hello World

Figure 2.7: Workspace and transcript window in Squeak

The above example was implemented using the object-oriented programming techniques available in SmallTalk. To implement the example using AOP technique one must make sure that the AspectS plug-in is installed. The installation link is available at [9].

2.3.1.1 Aspects

In AspectS, aspects are units of modularity that represent implementations of crosscutting concerns. They are identical to regular classes of SmallTalk. For example, an aspect *BeforeHelloWorld* can be created in the same category of *Test* class discussed in the previous section. The only difference is that they are defined as a subclass of the class *AsAspect* as shown in Listing 2.10.

[4] Workspaces are useful for typing snippets of Smalltalk code to experiment with. It can be also used for typing arbitrarily text to remember, such as to-do lists or instructions for anyone who will use the same image.

[5] The transcript is an object that is often used for logging system messages. It is a kind of "system console".

Listing 2.10 *BeforeHelloWorld* Aspect in AspectS

```
AsAspect subclass: #BeforeHelloWorld
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'HelloWorld'
```

Since AspectS is implemented without changing the SmallTalk syntax and virtual machine, the language features are the same for both AspectS and SmallTalk. For example:

- Aspects can extend classes and aspects.
- As Smalltalk does not support interface, aspects cannot implement interface.
- Aspects cannot be embedded inside classes as nested aspects.
- In AspectS, an aspect is installed by sending an install message to the instance of the aspect. An installed advice can be deactivated by sending the uninstall message to the same aspect instance.
- Aspects can be directly instantiated.

The weaving process in AspectS happens by sending an install message to the aspect instance (Figure 2.8). For unweaving or reversing the effect of aspect installation, the uninstall message is to be sent the same aspect instance. Weaving and unweaving in AspectS can be characterized as completely dynamic since it happens at runtime.

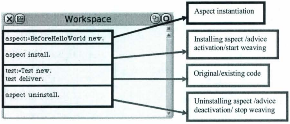


Figure 2.8: Dynamic weaving in AspectS

2.3.1.2 Join Point

Method execution is the only join point supported by AspectS. It can be defined as an object of *AsJoinPointDescriptor* class (shown in Listing 2.11) or a subclass of it. The class *AsJoinPointDescriptor* contains a static method *targetClass:targetSelector:* that specifies the method to be selected as join point.

Listing 2.11 AsJoinPointDescriptor Class in AspectS

```
Object subclass: #AsJoinPointDescriptor
 instanceVariableNames: 'targetClass targetSelector'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'AspectS-Aspects'
```

2.3.1.3 Pointcut

A pointcut in AspectS is a set of join points. It can be assigned to a variable which can then be passed as a method parameter. For example in Listing 2.12, *jpset1* is a pointcut that selects the deliver method of Test class as join point.

Listing 2.12 Pointcut in AspectS

```
|jpset1|
jpset1:=[{AsJoinPointDescriptor  targetClass:Test
                                 targetSelector: #deliver}].
```

2.3.1.4 Advice

An advice in AspectS is an object of the AsAdvice class. This advice object can be defined in the Squeak workspace and passed as a method parameter. Despite this, the most common way to define an advice object is to define it within a special method. The method is special because the name of the method starts with the prefix "advice" and the method returns an advice object (object of any of the subclasses of *AsAdvice*). The subclasses of *AsAdvice* are: AsBeforeAfterAdvice, AsAroundAdvice, AsHandlerAdvice and AsIntroductionAdvice. AspectS allows defining five different types of advices using these subclasses of AsAdvice. Usually each advice object is composed of the following components:

- Advice qualifier: an object of type AsAdviceQualifier which allows the description of dynamic attributes of a pointcut related to an advice. Section 2.3.1.5 provides a brief discussion on different advice qualifier attributes.
- Pointcut: a set of join points or a pointcut object.
- Block context: a code block which contains the crosscutting behavior along with the context information from join points.

a) AsBeforeAfterAdvice

An object of *AsBeforeAfterAdvice* allows adding behavior before the join point, after the join point, or both before and after the join point.

Listing 2.13 *adviceBefore* method of *BeforeHelloWorld* Aspect

```
adviceBefore
^ AsBeforeAfterAdvice
 qualifier: (AsAdviceQualifier
            attributes: {#receiverClassSpecific})
 pointcut: [{AsJoinPointDescriptor
            targetClass: Test targetSelector: #deliver}]
 beforeBlock: [:receiver :arguments :aspect :client |    Transcript
show: '  "Hello World   '.]
```

The method *adviceBefore,* as shown in Listing 2.13, returns an object of *AsBeforeAfterAdvice*. In the advice object the presence of a before block indicates that this advice will place the message "Hello World" before the execution *deliver* method. The same type of object can also be used to add behavior after a certain join point by replacing the *beforeBlock* with an *afterBlock*. The method *adviceAfter* in Listing 2.14 returns the object of type *AsBeforeAfterAdvice* which has an *afterBlock* in it. As a result, this advice adds the message "Goodbye World" after the execution of *deliver* method.

Listing 2.14 *adviceAfter* method of *AfterHelloWorld* Aspect

```
adviceAfter
  ^ AsBeforeAfterAdvice
   qualifier: (AsAdviceQualifier attributes:
{#receiverClassSpecific})
   pointcut: [{AsJoinPointDescriptor
             targetClass: Test targetSelector: #deliver}]
   afterBlock: [:receiver :arguments :aspect :client :return|
             Transcript show: '  Goodbye World.  '.]! !
```

However, to add some behavior both before and after the join point both *beforeBlock* and *afterBlock* should be used within the same advice object of *AsBeforeAfterAdvice* type.

b) AsAroundAdvice

The object of type *AsAroundAdvice* is used to modify behavior around a method execution. The *aroundBlock* within the object holds the information of behavior to be placed around the join point. The method *adviceAround* in Listing 2.15 returns the object of type *AsAroundAdvice*. This advice replaces the existing behavior of *deliver* method with the message "Welcome to AspectS".

Listing 2.15 ***adviceAround*** **method of** ***AroundHelloWorld*** **Aspect**

```
adviceAround
  ^ AsAroundAdvice
    qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific})
    pointcut: [{AsJoinPointDescriptor
                targetClass: Test targetSelector: #deliver}]
    aroundBlock: [:receiver :arguments :aspect :client :clientMethod |
                Transcript show: 'Welcome to AspectS' .]
```

c) AsIntroductionAdvice

The object of *AsIntroductionAdvice* is used to introduce new behavior in the program. For example, let us assume that we have an aspect named *IntroHelloWorld* under the category *HelloWorld*. The method *adviceIntro* of *IntroHelloWorld* as shown in Listing 2.16 returns an object of *AsIntroductionAdvice* to introduce a new method named *deliver2* to a class named *Test*. The methods' body is defined inside the *introBlock*.

Listing 2.16 ***adviceIntro*** **method of** ***IntroHelloWorld*** **Aspect**

```
adviceIntro
  ^ AsIntroductionAdvice
    qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific})
    pointcut: [{AsJoinPointDescriptor
                targetClass: Test targetSelector: #deliver2}]
    introBlock: [:receiver :arguments :aspect :client |
                Transcript show: 'Hello Intro'.]
```

Figure 2.9 shows that if we enable the *IntroHelloWorld* aspect, the method *deliver2* is created as it is introduced by the *AsIntroductionAdvice*. As a proof, if we call *deliver2*, the execution of this method produces the output "Hello Intro" in the transcript. However, if we disable the aspect the *AsIntroductionAdvice* is no more in effect. As a result, the call to the method *deliver2* ends up producing an error.

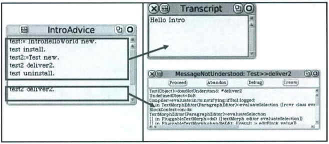


Figure 2.9: Testing a method introduction using AsIntroductionAdvice

Like the above example, within a same *AsIntroductionAdvice* object multiple methods to same or different class can be introduced. *However,* because each *AsIntroductionAdvice* object consists of a single *introBlock* that defines the body of the method or methods to be introduced; if multiple methods are introduced using the same *AsIntroductionAdvice* object, all methods will have the same body.

d) AsHandlerAdvice

An object of *AsHandlerAdvice* allows placing an exception handler around the sending of a message. Besides having the components similar to the previous advices, it includes an additional component to specify an exception class. An exception handler block is executed only when the sending of the message results in signaling such an exception [6]. For example, assume that *AspectHandler* aspect is having a method named *adviceException* as shown in Listing 2.17. The *AsHandlerAdvice* object includes *Error* class as an exception. Whenever an error of type *Error* is signaled, an exception is raised and handled by the handler block.

Listing 2.17 *adviceException* method of *AspectHandler* Aspect

```
adviceException
  ^AsHandlerAdvice
    qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific})
    pointcut: [{AsJoinPointDescriptor
                targetClass: TestHandler targetSelector: #deliver}]
    exception: Error
    handlerBlock: [:receiver :arguments :aspect :client :ex |ex signal.
                Transcript show: 'Exception Handled'.].
```

2.3.1.5 Advice Qualifier

AsAdviceQualifier has a class method named *attributes which* takes a set of advice qualifier attributes. Advice qualifier attributes are grouped roughly into the following two categories: Sender/ receiver aware activation and Cflow activation [6].

Sender/receiver aware activation can be further classified as receiver-class-specific, receiver-instance-specific, sender-class-specific, and sender-instance-specific. On the other hand, cflow activation can be further classified as class-first, class-all-but-first, instance-first, instance-all-but-first, super-first, and super-all-but-first.

In an advice, these attributes should be specified with certain constraints. First, attributes must be valid. Second, a set of attributes cannot be empty. Finally, at most one sender/receiver attribute must be present in each attribute set.

a) Sender/receiver aware activation

i. Receiver Class Specific:

A receiver-class-specific advice affects all receivers of the message that are an instance of a certain class [6]. For example, in Listing 2.14, *Test* class was the receiver. As shown in Figure 2.10, both the instances *test* and *test1* of *Test* class receive the message *deliver*. Since *deliver* was the target selector of the receiver class specific advice shown in Listing 2.14, both of these instances are affected by this advice.

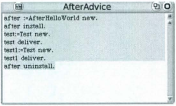


Figure 2.10: Weaving of AfterHelloWorld Aspect

ii. Receiver Instance Specific:

A receiver-instance-specific advice affects certain receivers of the message that are an instance of a certain class. Moreover, instances of prospective receivers should be added to or removed from the advice's aspect [6]. For example the advice in Listing 2.18, is a receiver-instance-specific advice. Although here the target class and target selector are similar to the previous example, this time only specific instances that are added as the receiver of the advice's aspect are affected by the advice.

Listing 2.18 *adviceAfter* method of *AfterHelloWorldRIS* Aspect

```
adviceAfter
  ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier attributes:
{#receiverInstanceSpecific})
    pointcut: [{AsJoinPointDescriptor targetClass: Test targetSelector:
#deliver}]
    afterBlock: [:receiver :arguments :aspect :client :return|

                    Transcript show: '  Goodbye World.  '.]
```

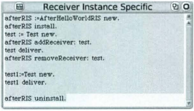


Figure 2.11: Weaving of AfterHelloWorldRIS Aspect

As shown in Figure 2.11, both the instances *test* and *test1* of *Test* class receive the message *deliver*. However, the receiver instance specific advice affects only *test*, since it is added as the receiver of *AfterHelloWorldRIS* aspect.

iii. Sender Class Specific:

A sender-class-specific advice qualifier attribute can be specified with the code shown below:

```
  qualifier: (AsAdviceQualifier attributes: {#senderClassSpecific})
```

If the qualifier in Listing 2.19 is replaced with this code, the target class *Test* will be affected only if the sender is of a certain class or its subclasses. Moreover, sender classes should be added to or removed from the advice's aspect [6].

For example, as shown in Listing 2.19, *test1* of *Test* class is created within the body of method *newMethod* of *NewTest* class and then the message *deliver* is sent to *test1*. As a result, *NewTest* can be considered as a sender class.

Listing 2.19 *newMethod* method of *NewTest* Class

```
newMethod
  |test1|
  test1:=Test new.
  test1 deliver.
```

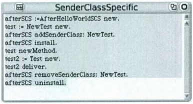


Figure 2.12: Weaving of AfterHelloWorldSCS Aspect

Now, as shown in Figure 2.12, the execution of deliver method is advised only when *NewTest* is added as a sender class and an instance of *NewTest* receives the message *newMethod*. Even though *test2* an instance of *Test class* receives the same message *deliver*, it is not affected with the advice since this time *NewTest* class is not the sender.

iv. Sender Instance Specific

A sender-instance–specific advice affects the receivers of messages that are instances of a certain class if senders are some specific instances of a certain sender class. Moreover, prospective senders are to be added to or removed from the advice's aspect [6].

```
qualifier: (AsAdviceQualifier attributes: {#senderInstanceSpecific})
```

For example, if the qualifier in Listing 2.19 is replaced with the above mentioned code, it will work as a sender-instance–specific advice. As *test,* a particular instance of *NewTest* Class (shown in Figure 2.13) is added as a sender of *AfterHelloWorldSIS* aspect, the advice affects the receivers only when the sender is *test,* but not the other instances of the same sender class.

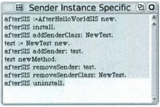


Figure 2.13: Weaving of AfterHelloWorldSIS Aspect

b) CFlow activation

In AspectS, with a cflow advice, the activation test examines the base context chain (Smalltalk's stack) for one of the following conditions depending on the type of cflow attribute specified in the advice qualifier attribute set [6]:

- one or more senders with the same class as the receiver
- one or more appearances of the receiver instance in it
- a send of the current message to super

The following is an implementation of a factorial example with object recursions to examine and understand different types of cflow advices:

The base class *AsFactorialM* for this factorial example is shown in Listing 2.20. It consists of an instance variable and three instance level methods.

Listing 2.20 *AsFactorialM* Class

```
Object subclass: #AsFactorialM
  instanceVariableNames: 'other'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ModifiedFactorial'
```

The method *initialize:* shown in Listing 2.21, receives an object or instance of *AsFactorialM* class and sets the value of instance variable *other* with the object.

Listing 2.21 *initialize* method of *AsFactorialM* Class

```
initialize: asFact
  other:=asFact.
```

The method *other:* shown in Listing 2.22, whenever called, works like the previous method.

Listing 2.22 *other* method of *AsFactorialM* Class

```
other: fact
  other :=fact.
```

The method *factorial:* shown in Listing 2.23 recursively calls itself and finally returns an integer if and only if the argument passed to it is not a negative number.

Listing 2.23 *factorial* method of *AsFactorialM* Class

```
factorial: anInteger
    anInteger = 0 ifTrue: [^ 1. ].
  anInteger > 0 ifTrue: [^ anInteger * (other factorial:
anInteger - 1).].
  self error: 'Not valid for negative integers.'.
```

To trace the above program, we create an aspect named *Trace* containing a method *adviceTrace* (as shown in Listing 2.24). Method *adviceTrace* returns a receiver-class-specific advice which is activated with the execution of *factorial:* method of *AsFactorialM* class. This advice does not contain any of the advice qualifier attributes of cflow type but helps to trace the method execution each time with its parameter and returned value.

Listing 2.24 *adviceTrace* method of *Trace* Aspect

```
adviceTrace

^ AsBeforeAfterAdvice
   qualifier: (AsAdviceQualifier
      attributes: { #receiverClassSpecific. })
   pointcut: [{ AsJoinPointDescriptor
      targetClass: AsFactorialM targetSelector: #factorial:. }]
   beforeBlock: [:receiver :arguments :aspect :client |
      Transcript show: 'Execution with: '.
      Transcript show: arguments;printString; cr.]
   afterBlock: [:receiver :arguments :aspect :client :return |
      Transcript show: 'Exit with: '.
      Transcript show: return;printString; cr.]
```

The codes shown on the left part of Figure 2.14 are the codes to be written in the workspace. We create two instances *fact1* and *fact2* of *AsFactorialM* class. Using the method *initialize:* and *other*: we switch over between these two instances while calling the *factorial:* method recursively. The reason behind using the two instances is to examine the differences between cflow attributes in later sections. The right part of Figure 2.14 shows the output produced in transcript after executing the code in the workspace.

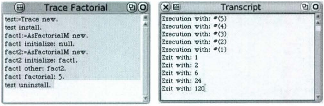


Figure 2.14: Workspace code and output of the factorial example

*i. **Class First***

A class-first advice is activated on an object-recursion's first method invocation [6]. If the advice qualifier attribute set of Listing 2.24 is replaced with the following code, the advice will be working as a class-first advice.

```
qualifier: (AsAdviceQualifier
      attributes: { #receiverClassSpecific.#cfFirstClass. })
```

This class-first advice will be triggered only when the method *factorial:* is invoked for the first time by the instance of *AsFactorialM* class. It is to be noted that this method invocation is class dependent. As a result, no matter how many instances are used to invoke the method, as shown in Figure 2.15, only the first invocation will be advised.

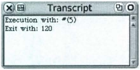


Figure 2.15: Transcript for Class First Attribute Example

ii. Class All-But-First

A class-all-but-first advice will trigger activation on object-recursions other than the first method invocation [6]. If the advice qualifier attributes set of Listing 2.24 is replaced with the following code, the advice will be working as a class-all-but-first advice.

```
qualifier: (AsAdviceQualifier
attributes: { #receiverClassSpecific.#cfAllButFirstClass. })
```

Class-all-but-first advice will be triggered every time except for the first time when the method *factorial:* is invoked by the instance of *AsFactorialM* class. As this method invocation is class dependent, no matter how many instances are used to invoke the method, as shown in Figure 2.16, all the invocations except the first one will be advised by this advice.

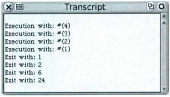


Figure 2.16: Transcript for Class All But First Attribute Example

iii. Instance First

An instance-first advice will trigger activation on a method-recursion's first method invocation [6]. If the advice qualifier attributes set of Listing 2.24 is replaced with the following code, the advice will be working as an instance-first advice.

```
qualifier: (AsAdviceQualifier
attributes: { #receiverClassSpecific.#cfFirstInstance. })
```

For the factorial example, an instance-first advice will be triggered only when the method *factorial:* is invoked for the first time by the instances of *AsFactorialM* class. Since this method invocation is instance dependent, each instance will be considered separately. Moreover, in this example, as we switch between instances of *AsFactorialM* class, the first invocation by *fact1* with the argument 5 and the first invocation by *fact2* with the argument 4 will be advised by this instance-first advice. Figure 2.17, shows the output produced in the transcripts after installing an aspect with instance-first advice.

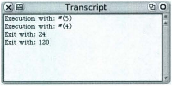


Figure 2.17: Transcript for Instance First Attribute Example

iv. Instance All-But-First

An instance-all-but-first advice will trigger activation on a method-recursion other than the first method invocation [6]. If the advice qualifier attributes set of Listing 2.24, is replaced with the following code, the advice will be working as an instance-all-but-first advice.

```
qualifier: (AsAdviceQualifier
attributes:{ #receiverClassSpecific.#cfAllButFirstInstance. })
```

For the factorial example, an instance-all-but-first advice will be triggered every time except when the method *factorial:* is invoked for the first time for each instance of *AsFactorialM* class. Since this method invocation is also instance dependent, each instance here will be also considered separately. As a result, except for the first invocation by *fact1* with the argument 5 and the first invocation by *fact2* with the argument 4, the other invocations will be advised by this instance-all-but-first advice. Figure 2.18 shows the output produced in the transcripts after installing an aspect with instance-all-but-first advice.

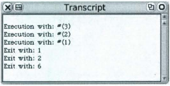


Figure 2.18: Transcript for Instance First Attribute Example

v. Super First

To activate the super-first cflow advice, the Smalltalk stack is examined for a send of a current message to the super class. The advice is activated if there is no such send [6]. For example we create a new class *FactorialM* with a method *factorialM:* as shown in Listing 2.25. Also, we create a subclass *SubFactorialM* of *FactorialM* (shown in Listing 2.26).

Listing 2.25 *factorialM* method of *FactorialM* Class

```
factorialM: anInteger

 Transcript show: 'factorialM method of FactorialM class';cr.
 anInteger <= 1 ifTrue: [^ 1. ].
 anInteger > 1 ifTrue: [^ anInteger * (self factorialM:
anInteger - 1).].
 self error: 'Not valid for negative integers.'.
```

Listing 2.26 SubFactorialM Class

```
FactorialM subclass: #SubFactorialM
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''
 category: 'ModifiedFactorial'
```

The class *SubFactorialM* also possesses a method named *factorialM:* (Listing 2.27) from where the method *factorialM:* of super class (*FactorialM*) is called.

Listing 2.27 factorialM method of SubFactorialM Class

```
factorialM: anInteger

 Transcript show: 'factorialM method of SubFactorialM class';cr.
 anInteger <= 1 ifTrue: [^ 1. ].
 anInteger > 1 ifTrue: [^ anInteger * (super factorialM: anInteger
- 1).].
 self error: 'Not valid for negative integers.'.
```

Listing 2.28 shows an advice method of *AspectSuperFirstM* Aspect which returns a super-first cflow advice. Moreover, the method *factorialM:* of super class i.e. *FactorialM* is specified as the join point in this advice.

Listing 2.28 ***adviceTrace*** **method of** ***AspectSuperFirstM*** **Aspect**

```
adviceSuperFirstM
  ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier
       attributes: { #receiverClassSpecific.#cfFirstSuper})
    pointcut: [{ AsJoinPointDescriptor
       targetClass: FactorialM targetSelector: #factorialM:. }]
    beforeBlock: [:receiver :arguments :aspect :client |
     Transcript show: 'Execution with: '.
     Transcript show: arguments;printString; cr.]
    afterBlock: [:receiver :arguments :aspect :client :return |
     Transcript show: 'Exit with: '.
     Transcript show: return;printString; cr.]
```

To test the above advice, we execute the following code as shown in the left side of Figure 2.19. Since the Smalltalk stack is examined for the sending of a current message *factorialM:* to the super (*FactorialM*) and such sending is found, super-first cflow advice remains inactive. As a result we can only see some test messages from the methods generated in the Transcript (right side of Figure 2.19).

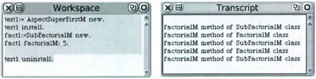


Figure 2.19: Workspace and Transcript for Super First Attribute Example

vi. Super All-But-First

To activate the super-all-but-first cflow advice, the Smalltalk stack is examined for sending of a current message to the super. The advice is activated if such a message is found [6]. For example, if the qualifier in advice method shown in Listing 2.28, is modified with the following codes, the advice will work as a super-all-but-first cflow advice.

```
qualifier: (AsAdviceQualifier
      attributes: { #receiverClassSpecific.#cfAllButFirstSuper})
```

Now, the same code that was shown in the workspace for the previous example, if executed, whenever the factorialM: of FactorialM class is called, super-all-but-first cflow advice will be activated as shown in Figure 2.20.

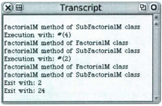


Figure 2.20: Transcript for Super All But First Attribute Example

The next section implements our ShoppingCart running example in AspectS.

2.3.2 Running Example in AspectS

While implementing the Shopping-Cart running example in AspectS, the base model is kept without any logging code. The base model consists of four classes: *AsItem*, *AsInventory*, *AsShoppingCart* and *AsShoppingCartOperator*.

The *AsItem* class models the items that can be purchased. This class has a constructor method *initialize:withprice:* and three public methods: *getId:*, *getPrice:* and *toString:*. The *getId:* and the *getPrice:* methods provide the identifier and price of the item respectively. The *toString:* method simply sets the format of an item to a string. Id and price of items are initialized and stored respectively in the instance variables *id* and *price*.

The *AsInventory* class represents the list of items available for purchasing. This class has two public methods: *addItem:* and *removeItem:*. Both of these methods take an item as argument which can be added to or removed from the existing item inventory using these two methods. The list of items in the inventory is represented using the instance variable *listItems*.

The *AsShoppingCart* class represents the list of items in a shopping cart of a customer. The two public methods *addItem()* and *removeItem()* are respectively used to add and delete specific items from the shopping cart's item list maintained by the instance variable *listItems*.

The *AsShoppingCartOperator* class is used to model the operations related to a purchase. In the original example [1] this class has two static public methods to update both the item lists of inventory and shopping cart. Since in AspectS static or class level methods cannot be advised, we implement these two methods as regular or instance level methods. The method *shoppingCart:inventory:addItem:* takes an instance of *AsShoppingCart*, an instance of *AsInventory*, and the item to be added to the ShoppingCart list. The purpose of this method is to model a purchase of an item by adding an item to the shopping cart and deleting the same item from the inventory. On the other hand *shoppingCart:inventory:removeItem:* takes an instance of *AsShoppingCart*, an instance of *AsInventory* and the item to be removed from the ShoppingCart list. This method is used to model a return of an item by removing an item from the shopping cart and adding it back to the inventory.

Listing 2.29 *adviceTrace* method of *AspectSuperFirstM* Aspect

```
adviceLogging

   ^ AsBeforeAfterAdvice
   qualifier: (AsAdviceQualifier attributes:
{#receiverClassSpecific})
   pointcut: [{AsJoinPointDescriptor
        targetClass: AsInventory targetSelector: #addItem:.
                     AsJoinPointDescriptor
        targetClass: AsInventory targetSelector: #removeItem:.
                     AsJoinPointDescriptor
        targetClass: AsShoppingCart targetSelector: #addItem:.
                     AsJoinPointDescriptor
        targetClass: AsShoppingCart targetSelector: #removeItem:.
                     AsJoinPointDescriptor
        targetClass: AsShoppingCartOperator targetSelector:
 #shoppingCart:inventory:addItem:.
                     AsJoinPointDescriptor
        targetClass: AsShoppingCartOperator targetSelector:
#shoppingCart:inventory:removeItem:.}]
   beforeBlock: [:receiver :arguments :aspect :client |
            Transcript show: (Date today) printString,
                    ' ',
                    (Time now) printString,
                    ' ',
                    ( receiver class ) printString;cr.
            Transcript show: '  INFO: Entering  ';cr.]
```

The crosscutting-concerns for this example are implemented using an aspect named *AspectTrace*. The advice method named *adviceLogging,* shown in Listing 2.29, returns a receiver-class-specific advice that places some additional behaviors or logging codes before and after each join point.

To test the entire example in workspace (Figure 2.21), we create *demoAspect,* an instance of *AspectTrace,* and then send the install message to it. Next, an instance of AsInventory class is created. This instance is used to add three items in the inventory list. Subsequently, the instances of *AsShoppingCart* and *AsShoppingCartOperator* are created. After that, the instances of *AsInventory* and *AsShoppingCart* along with the items to be added to the ShoppingCart are passed to the methods of *AsShoppingCartOperator* class. Lastly, the aspect is uninstalled.

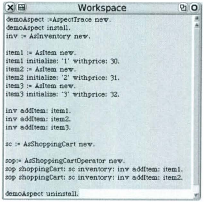


Figure 2.21: Workspace codes for ShoppingCart example in AspectS

The execution of the above code in the workspace places the logging codes before and after each join point mentioned in the advice of *AspectTrace*. The output in the transcript is shown in Figure 2.22.

Figure 2.22: Output for ShoppingCart example in AspectS

2.4 AspectML

2.4.1 An Overview of AspectML

AspectML is a typed, functional, aspect-oriented programming language based on ML [8]. Besides providing aspect-oriented programming language features, AspectML provides run-time type analysis and seamless integration of polymorphism. The syntax of AspectML is an extension of the syntax of idealized AspectML [8] with many common constructs following Standard ML. Anyone who is familiar with the language ML and has some experience with at least one aspect-oriented programming language can easily work with AspectML. Since there are syntactical differences between ML and AspectML, it is worth starting with some AspectML examples to get familiar with the language.

Example 1: Creating an empty list

In ML we can create an empty list as follows:

```
-val a=[];
```

ML responds with:

```
val a=[]:'a list
```

But due to syntactical issue, in AspectML the same code ends up with an uncaught exception error. One solution to that problem could be defining a function which takes an empty list and create an empty list and then keep the result of the function call in a variable as follows:

```
%fun emptyList ([])=[];
val c=emptyList([]);
```

In the above code, a function *emptyList* takes an empty list and produces an empty list as well. The result of the function call is an empty list which is kept in the variable "c".

Example 2: Creating tuples

In ML creating a new type is very simple. We can create a type "Item" which is a tuple of string and integer as follows in ML:

```
-type Item=string*int;
```

The above type Item then can be used to crate items as shown below. In the following code, item1, item2, and item3 are three different items of type Item. Each of these items is a tuple containing a string and an integer.

```
-val item1:Item=("1",30);
-val item2:Item=("2",31);
-val item3:Item=("3",32);
```

Unlike ML, we can't create a type in AspectML. However, tuples can be created by using functions like *createItem,* which takes a string and an integer as argument and produces a tuple. The following code presents the function and its uses to create items using AspectML constructs:

```
%fun createItem(id:String, price:Int)=(id, price);
%val item1=createItem ("1",30);
%val item2=createItem ("2",31);
%val item3=createItem ("3",32);
```

Example 3: Creating a list of tuples

In AspectML a list of tuples can be created as follows:

```
%val a=[("1",30), ("2",31),("3",32)];
```

The following output indicates that a list of tuple "a" has been created. It provides detail information of the type of tuples along with its elements.

Output

```
val a=(( : : ("1",30))(( : :  ("2",31)) (( : : ( "3",32))[]) ) )
val a<= ::[(Tuple (TCons String) (TCons Int) TNil) ) )]("1",30)::
[(Tuple (TCons String) (TCons Int) TNil) ) )]("2",31):: [(Tuple
(TCons String) (TCons Int) TNil) ) )]("3",32)[] [(Tuple (TCons
String) (TCons Int) TNil)))]
```

Example 4: Adding a new element to the existing list

In AspectML a new element can be added to an existing list. For example we can add an element that is a tuple of a string and an integer to the list created in previous example as follows:

```
%val b=("4",33):: a;
```

Output

```
val b=(( : : ( "4",33))a)
val b<=::[(Tuple (TCons String) (TCons Int) TNil) )
)]("4",33)::[(Tuple (TCons String) (TCons Int) TNil) )]("1",30)::
[(Tuple (TCons String) (TCons Int) TNil) )]("2",31):: [(Tuple
(TCons String) (TCons Int) TNil) )]("3",32)[] [(Tuple (TCons
String) (TCons Int) TNil)))]
```

In the above code, "b" is a new list containing the element ("4",33). The other elements from previous list "a" are also present in "b".

AspectML does not have a concept known as Aspect. Besides having some syntactical differences to ML, AspectML has special language constructs for advices consisting of the body and the pointcut designators.

2.4.1.1 Join Point

In AspectML, the function call is the only exposed join point.

2.4.1.2 Pointcut

In AspectML a *pointcut designator* has two parts: a *trigger time*, which may either be *before*, *after*, or *around* and a *pointcut proper*, which is a set of function names. Pointcuts identify join points in the program flow. However, unlike AspectJ, not the pointcut but the advices can expose the context at the matched join point. In AspectML, pointcuts do not have names. The join points are either described by a set of function names or by using the keyword *any*. The point cut designator **before (|#f#|)** represents the point in time immediately before executing call to the function **f**. Likewise, the point cut designator **after (|#f#|)** represents the point in time immediately after execution of the function f. The pointcut designator **around (|#f#|)** wraps around the execution of a call to the function f [8].

2.4.1.3 Advice

Advice in AspectML includes two parts, the body, which specifies what to do, and the pointcut designator, which specifies when to do it [8]. An advice does not have a name and cannot be called directly (it is the system's job to execute it). It does not have an access specifier. An advice can capture the method's context, such as the method's arguments. AspectML allows defining type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points [8]. The application of advice in AspectML usually varies with the *trigger time* (before after, or around) of pointcut designators.

2.4.2 Running Example in AspectML

Since AspectML is a functional language, there is no concept of classes in it. However, it is possible to implement the methods of ShoppingCart example as functions of AspectML. In Section 2.4.1, we have seen that how list of tuples can be created using AspectML constructs. Here also, in Listing 2.30, we create a function named *createItem* that takes id and price and produces a tuple of id and price. According to the function definition, the datatypes of id and prices should be String and Int, respectively. Furthermore, each tuple produced using the function *createItem* will be considered as an item for this example.

Listing 2.30 Creating items in AspectML

```
fun createItem(id:String, price:Int)=(id,
price);
val item1=createItem ("1",30);
val item2=createItem ("2",31);
val item3=createItem ("3",32);
```

Listing 2.31 shows two function *invAddItem* and *invRemoveItem* related to the Inventory of ShoppingCart example. *invAddItem* takes a list and an item, and produces a new list by adding the item in it. On the other hand, the purpose of *invRemoveItem* is to remove an item from the Inventory. It takes the existing inventory list and an item, and produces a new list without that item.

Listing 2.31 Functions related to Inventory

```
fun invAddItem(invList, item)=(item::invList);
fun invRemoveItem(invList,item)=
  case (invList)
    |([])=>([])
    |(hdinvList::tlinvList)=>
      if (hdinvList==item) then(tlinvList)
      else
       let
        val (invlistNew)=(invRemoveItem( tlinvList, item))
       in
        (hdinvList::invlistNew)
       end;
```

In the above listing, *invList* is an existing List. An item can be added to *invList* as shown in the code. However, while removing an item from the existing list, we have to check several cases. That is why, within the function *invRemoveItem*, we added case expression and conditional statements. At first, we had to check whether the list is empty or not. If the list is empty, there is nothing to be removed from it. Other wise we divide the list into head and tail. Then each item in the list head is checked with the item to be removed. Whenever the item to be removed is matched with an item in the list, it is removed from the list.

Listing 2.32 shows two functions *cartAddItem* and *cartRemoveItem* related to the ShoppingCart of our running example. The purpose of these two functions is similar to that of the functions related to inventory. However, the list to be used in these two functions is the item list of ShoppingCart, not the Inventory. The function *cartAddItem* takes a list and an item, and produces a new list by adding the item in it. *cartRemoveItem* takes the existing cart item list and an item, and produces a new list without that item. Similar to Listing 2.31, in Listing 2.32 we have used case statements and conditional statement.

Listing 2.32 Functions related to Shopping Cart

```
fun cartAddItem(cartList, item)=(item::cartList);

fun cartRemoveItem(cartList,item)=
  case (cartList)
    |([])=>([])
    |(hdcartList::tlcartList)=>
      if (hdcartList==item) then(tlcartList)
      else
       let
        val (cartlistNew)=
           (cartRemoveItem(tlcartList, item))
       in
        (hdcartList::cartlistNew)
       end;
```

Listing 2.33 shows one of the functions related to the Shopping Cart Operator of our running example. This function takes an inventory list, a cart list and an item as arguments. In order to create a new inventory list and a cart list, this function uses *invRemoveItem* to remove the item from the existing inventory list and *cartAddItem* to add the item to the existing cart list. Since we will be using only this function of Shopping Cart Operator, the other function, *cartOperatorRemoveItem,* is not shown here.

Listing 2.33 A Function of ShoppingCartOperator

```
fun cartOperatorAddItem(invList, cartList, item)=
        let
          val (invlistNew)=invRemoveItem(invList,item)
          val (cartlistNew)=cartAddItem(cartList,item)
        in
          (invlistNew,cartlistNew)
        end;
```

The previous four Listings (from Listing 2.30 to Listing 2.33) present the base model for the ShoppingCart example in AspectML.

Listing 2.34 shows the advices in AspectML for the ShoppingCart example. Since AspectML does not allow using the same advice for functions with different data types, we use two before advices for two different categories of functions. In the first advice, the functions *invAddItem*, *invRemoveItem,* and *cartAddItem* are mentioned as the join points. In the second advice, the function *cartOperatorAddItem* is mentioned as the join point.

Listing 2.34 advices in AspectML for ShoppingCart example

```
advice before (| #invAddItem,invRemoveItem,cartAddItem# |) (arg, s,
info) = (println"  ";print "INFO: Entering "; print("   "^
(getFunName info)); arg )
advice before (| #cartOperatorAddItem# |) (arg, s, info) =
(println"  ";print "INFO: Entering "; print("   "^ (getFunName
info)); arg )
```

Listing 2.35 shows the code to test the functionality of the above advices along with the codes of base model. First, three items, one at a time, are added to the inventory list. Then the function *cartOperatorAddItem* is called to remove the item2 from the existing inventory list and to add to the cart list

Listing 2.35 Codes to test the program

```
let
  val (newInvList) = invAddItem([], item1)
  val (newInvList1)=invAddItem(newInvList, item2)
  val (newInvList2)=invAddItem(newInvList1, item3)
  val (newInvList3,newCartList1)=cartOperatorAddItem(newInvList2,
[], item1)
in
  cartOperatorAddItem(newInvList3, newCartList1, item2)
end;
```

Figure 2.23 shows the output produced after compiling the above codes in AspectML compiler.

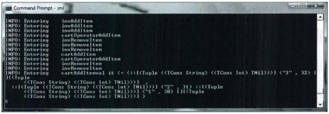


Figure 2.23: The advices of Listing 2.34 were triggered whenever the functions mentioned in the join points were called from Listing 2.35.

Chapter 3

AOP Approaches: Static and Dynamic

3.1 Static AOP

Static AOP, as implemented in AspectJ [3], requires the developer to specify all pointcuts, advice and aspects at compile time. Usually a weaving compiler is used to add advice code to join points. When several aspects match the same join point [15, 16], the aspects are woven in a statically-defined order. As a result, aspects cannot be added, removed, or modified at runtime [16]. To change aspects, the system must be recompiled [17].

The ShoppingCart example implemented with AspectJ in Section 2.2.2 complies with the static AOP approach, as the example does not allow us to start, stop, or modify the aspect configuration during runtime. However, a static language can approximate dynamic adaptation through run-time checks. Figure 3.1 follows the adaptation of the AspectJ example given in Section 2.2.2. Here, we have a user interface which comes with switches to turn the logging on and off. Using this interface one can press the "Start" button and see the program running without logging code. However, the logging feature will be enabled whenever "Start Logging" button is pressed. The logging feature can be disabled any time while the program continues to run.

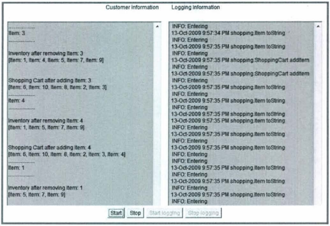


Figure 3.1: User interface for modified ShoppingCart program

Since printing of the logging messages are related to the activation and deactivation of aspects, it seems that the modified ShoppingCart example allows enabling and disabling the aspects during runtime. However, this is not a truly dynamic AOP system. This dynamic adaptation is accomplished through run-time checks.

The control methods (*setEnabled* in Listing 3.1) must be called from the base system, which requires the base system to be aware of the aspects. This causes an (however minimal) overhead of checking the configuration conditions. Furthermore, for more complex control and configuration requirements, the complexity of conditional expressions increases rapidly.

Listing 3.1 ***Aspect Enabling/Disabling at Runtime***

```
public aspect TraceAspect {
private  static boolean logAspect = false;

public static void setEnabled(boolean flag) {
    TraceAspect.logAspect = flag;
  }

pointcut isEnabled() : if(logAspect);
pointcut traceMethods() : execution(* *.*(..))
        && !within(TraceAspect)
        && !within(UserInterface)
        && !within(PrintThread);

    before() : traceMethods()&& isEnabled() {
    ......

    }
}
```

Static AOP is suitable for systems that can be reconfigured and updated by stopping and restarting. However, static AOP shows pitfalls for long running systems, where this is not an option [17].

Using the example of a coffee ordering system illustrated in [18] and a client server application presented in [19], [16] shows how static AOP approach is not suitable for the applications while dynamically adding and removing responsibilities to an object.

[20] describes a scenario taken from telecommunications, where corrective actions need to be performed in a flexible manner on the integrated system if the system was not initially deployed correctly.

[21], [22] and [23] were motivated by the need for dynamic adaptation of distributed systems at runtime where the systems need to be updated with the changing environment. Since all applications are required to be stopped during the stop and

restart of system software with static adaptation, static adaptation techniques are not always suitable [21]. Moreover, dynamic or fast reconfiguration of distributed applications is needed to handle several concerns such as fault-tolerance, data consistency, remote version updating, run-time maintenance, dynamic server lookup, or scalability [23]. To adapt with the changed environment, there should be some option to add or remove concerns on existing applications during runtime.

3.2 Dynamic AOP

Dynamic AOP provides support for controlling aspects at runtime. As implemented in AspectS and AspectML, dynamic AOP allows changes to aspects without restarting the program [16]. A run-time weaver is used to add advice code to the selected join points. However, dynamic AOP is different from dynamic weaving, which allows installing and uninstalling aspect. As AspectJ does not support instantiation of aspect, neither dynamic AOP nor dynamic weaving is allowed in it.

Both dynamic AOP and dynamic weaving have some advantages:

- It removes AOP overhead when aspects are not required, e.g. profiling or tracing aspects on a production system.
- It allows dynamic configuration of aspect behavior, e.g. switching from tracing to profiling, without resetting the state of the base systems.
- It allows aspect re-configuration depending on the state of the base system.
- It allows extensible and reusable aspect libraries.

The latter is a consequence of the typical implementation of dynamic AOP in which the core AOSD concepts are provided using the primary modularization concepts. The

AspectS example above shows how advice and join point descriptors are implemented as objects. Hence, they can be used to build generic class or object libraries. While many dynamic AOP approaches are implemented this way, the choice of dynamic or static AOP and providing AOP with or without language extensions, are independent.

Dynamic AOP is easier to implement in interpreted languages such as Smalltalk or ML, although dynamic AOP versions of AspectJ exists [16]. Figure 3.2 presents a user interface in AspectS environment. Although this user interface is similar to the one shown in Section 3.1 , it allows dynamic control of aspect behavior.

Figure 3.2: User interface for Dynamic ShoppingCart Program

The start button, if pressed, will run the base system without the logging functionality. However, the aspect is enabled and prints the logging information once the "Start Log" button is pressed. In Figure 3.3, we cannot see any logging messages for *Item4* since logging was inactive while adding the item to the cart. However, logging was activated when *Item9* was in process. As a result, we can see the logging messages for the classes *InventoryD* and *ShoppingCartD*.

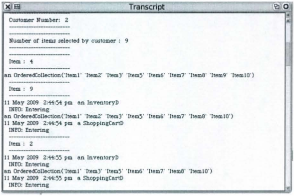


Figure 3.3: Output of dynamic ShoppingCart program

Although the static AspectJ implementation shown in Section 3.1 behaves exactly like the above program, the difference here is that the base system and aspect extensions can be enabled and disabled separately, e.g. from a separate control thread, as shown in the following four Listings (from Listing 3.2 to Listing 3.5). This also allows the reconfiguration of the aspect to adapt or configure the advice to change requirements without losing state of the base system.

Listing 3.2 Code for Start button

```
start
   |test1|
   process1 := [test1:=AsUserInterfaceD new.
   test1 run.] newProcess.
   self add1: 'Cart is Activated'.
   process1 resume.
```

Listing 3.3 Code for Stop button

```
stop
   self add1: 'Cart Disabled'.
   process1 terminate.
```

Listing 3.4 Code for Start Log button

```
startLog
    self add2: 'Logging Enabled'.
    demoAspect install.
```

Listing 3.5 Code for Stop Log button

```
stopLog
 self add2: 'Logging Disabled'.
 demoAspect uninstall.
```

Dynamic AOP frequently, but not necessarily, treats AOSD concepts as instances of the primary modularization concepts. For example, advice and pointcuts are objects in AspectS, and pointcuts are functions in AspectML. An AspectML example taken from [8] is shown in Listing 3.6. In this example, `toLog` of type `pc(<a b>a~>b)` is a pointcut, which is passed as an argument to the function *startLogger*.

Listing 3.6 Passing pointcut as argument

```
fun startLogger (toLog: pc(<a b> a~>b)) =
   let
   advice before (|toLog|) (arg, _, info) =
        ((print ("before : "^(getFunName info) ^ ":" ^
                         (val_to_string arg)^"\n")); arg)
   advice after (| toLog |) (res, _, info) =
        ((print ("after " ^ (getFunName info) ^ " : "^
                         (val_to_string res) ^ "\n"));res)
   in () end
```

Dynamic AOP allows us to build generic logging aspects that can be configured at runtime with the set of join points to be logged. For example, in the following codeListing 3.10), *AspectLogger* (Listing 3.7) is a generic logging subclass of *AsAspect*. It has a constructor method *newJP* (Listing 3.8) that allows initialization with a set of *AsJoinpointDescriptor* objects. These are stored by the aspect (Listing 3.9) and passed to the *adviceLogging* function (Listing 3.10), which is called by the run-time weaver when installing the aspect.

Listing 3.7 Class Definition of AspectLogger

```
AsAspect subclass: #AspectLogger
instanceVariableNames: 'jpset'
classVariableNames: ''
poolDictionaries: ''
category: 'First Class Pointcut'
```

Listing 3.8 Class method

```
newJP: aJPDescriptor
^(self new) jpset: aJPDescriptor; yourself.
```

Listing 3.9 Instance method

```
jpset: aJPDescriptor
jpset := aJPDescriptor.
```

Listing 3.10 The method adviceLogging

```
adviceLogging
^ AsBeforeAfterAdvice
qualifier: (AsAdviceQualifier
attributes:(#receiverClassSpecific))
pointcut:jpset
beforeBlock: [:receiver :arguments :aspect :client |
… logging code here … ]
afterBlock: [:receiver :arguments :aspect :client
:result |
… logging code here … ]
```

AspectJ does not provide instantiable[6] and configurable aspect, advice, or pointcut classes. It is instead based on language extensions handled by a weaving compiler. Hence, the above examples of generic and configurable aspects and advice cannot be implemented in AspectJ.

Recent work on dynamic AOP has focused on solving a number of issues and problems that are not well suited for static AOP implementations. Handi-Wrap is a dynamic AOP extension for Java which allows advice to be defined compositionally and supports run-time weaving [24]. PROSE *(PROgrammable extensionSions of sErvices)* is a dynamic AOP approach based on Java that allows aspects to be woven, unwoven, or replaced at run-time. PROSE supports rapid AOP prototyping and debugging and helps developers to understand the behavior of aspects in changed environment [25]. To address the recent demand for dynamic AOP, a new dynamic aspect weaver called Wool is presented in [26], which makes it possible to implement efficient dynamic AOP systems. Wool addresses the solution to the performance penalties caused in some prior implementations. An approach for language and platform independent dynamic AOP based upon reflection is presented in [22]. It focuses on dynamic adaptation of distributed systems at run-time. Dynamic AspectJ [16] considers the difficulties arising from the static scheduling strategy of AspectJ and shows how turning to a more dynamic strategy makes it possible to order, cancel, and deploy aspects at runtime.

[6] Although the user can specify how many aspect instances AspectJ is supposed to create, the user canot instantiate or free aspects at will.

Chapter 4

AOP LANGUAGE FEATURE COMPARISION

As discussed in Section 1.3.2.2 , Join Point Models (JPMs) of different AOP implementations can be compared based on the following criteria:

- Which join points are exposed,
- How pointcuts are specified,
- The operations permitted at the join points, and
- The structural enhancements that can be expressed.

Core JPM features e.g. method execution, exception raising or throwing are common across most AOP implementations. However, different languages provide concepts beyond these core features, such as the structural enhancements of AspectJ and AspectS. To cover a wide variety of JPM features, we examine:

- AspectJ – a static AOP approach,
- AspectS – a dynamic, object-oriented approach, and
- AspectML – a dynamic, functional approach to AOP.

AspectJ allows wide varieties of join point selections. Based on the JPM features of AspectJ, the following sections present a comparative picture of AspectJ, AspectS and AspectML.

4.1 AspectJ, AspectS and AspectML

4.1.1 Exposed Join Point Categories

Being a member of the functional language family, AspectML only exposes function calls as join points. However, the exposed join point categories for AspectJ and AspectS are not that simple. AspectJ exposes eight categories of join points, some of which can be found in AspectS too. To understand the difference between the exposed join point categories of AspectJ and AspectS, a detailed comparison is provided in the following sections.

a) Method join points

AspectJ exposes both method call and method execution as join points. The following code snippet shows a pointcut that selects execution of the *deliver* method of *Test* class in AspectJ.

```
pointcut deliverMessage()
   : execution (* Test.deliver(..));
```

Using AspectS, as shown below, the above example can be imitated by a receiverClassSpecific advice, which selects *Test* as the targetClass and method *deliver* as the targetSelector:

```
qualifier: (AsAdviceQualifier
    attributes: {#receiverClassSpecific})
pointcut: [{AsJoinPointDescriptor
    targetClass: Test targetSelector: #deliver}]
```

In AspectJ, a call pointcut is specific to a type signature. Thus, a call of a method will be selected as a join point if and only if the type of caller is matched with the type signature mentioned in the pointcut. For example, as shown in Listing 4.1, the method *deliver* is called by the instance of the *Test* class. The method *deliver* is called for the second time by an instance of *Interface1*. Accordingly, the method *deliver* is executed twice.

Listing 4.1 *Test* class

```
public class Test implements Interface1 {
    public void deliver() {
        System.out.println("In Test.deliver()");
    }
    public static void main(String[] args) {
        Test test = new Test();
        test.deliver();

        Interface1 interface1 = test;
        interface1.deliver();
    }
}
```

As shown in Listing 4.2, if the execution of the method *deliver* is selected by a pointcut as a join point and advised, the advice will be activated twice.

Listing 4.2 *TestAspect* aspect

```
public aspect TestAspect {
    before() : execution(void Test.deliver()) {
        System.out.println("Advising execution of Test.deliver()");
    }

    before() : call(void Test.deliver()) {
        System.out.println("Advising call of Test.deliver()");
    }
```

Although there were two calls to the *deliver* method, since the call pointcut specifies the type of a calling object, only the call from the instance of class *Test* is advised. Figure 4.1 shows the output of the above example.

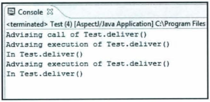


Figure 4.1: Output of the above example

AspectS does not provide any construct to select method call as join point. Hence, method execution is the only method join point in AspectS.

b) Constructor join points

Like Java/C++, object creation in AspectJ involves constructors. In AspectJ constructors are used to create and initialize new instances. AspectJ provides pointcut constructs to select both constructor call and constructor execution as join points. The execution of constructor is the constructor itself, e.g. the constructor of *Item* class shown in Listing 4.3.

Listing 4.3 Constructor in *Item* class

```
public class Item {
  public Item(String id, float price) {
    _id = id;
    _price = price;
  }
}
```

The call of constructor is the location of invocation of the constructor. For example, the following will be selected as a constructor call in AspectJ.

```
Item item1 = new Item("1", 30);
```

Since Smalltalk/AspectS does not have any special syntax or semantics for constructor [27], AspectS does not allow selecting the constructor call or execution as a join point. The functionality of the above code, written in Java, can be achieved by the following code snippet of SmallTalk.

```
item1 := AsItem new.
item1 initialize: '1' withprice: 30.
```

Here *item1,* an instance of *Item* class is created by using a class method *new.* Also, assigning the values of the fields' *item* and *price* does not need the involvement of constructor, since the values are assigned by a regular method of SmallTalk named *initialize:withprice:*.

c) Field access join points

The field access join points capture the read and write access to an instance or class member of a class [1]. AspectS does not provide any pointcut constructs to select an instance or class member of a class directly. However, if the fields are accessed using regular methods of SmallTalk, then field access can be advised by selecting the execution of those methods as join points. For example, in Listing 4.4, the method *n:* sets the value of the field *n.*

Listing 4.4 A setter method in AspectS

```
n: anInteger
  n := anInteger.
```

Listing 4.5 Advising the setter method *n:*

```
adviceSetField
  ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier
         attributes: {#receiverClassSpecific})
    pointcut: [{AsJoinPointDescriptor
         targetClass: AsCounterModified1 targetSelector:
#n:}]
    beforeBlock: [:receiver :arguments :aspect :client |
           Transcript show: '  "Set Field"   '.]
```

In AspectS, write access to the field *n* can be advised indirectly by capturing the setter method *n:* as join point (Listing 4.5).

d) Exception handler execution join points

Both AspectJ and AspectS supports selection of the exception handler execution join points. In AspectJ, an exception handler execution join point encompasses the catch block [1].

Listing 4.6 Exception handler execution join point

```
package exception;
public class TestHandler {
 public void deliver (){
     try {
         Error error= new Error("Error Occured");
         throw error;
     }
     catch (Error e) {
         System.out.println("Inside Catch block");
     }
 }
}
```

Listing 4.6 contains a handler block, which can be captured in AspectJ by the pointcut shown in Listing 4.7.

Listing 4.7 *AspectHandler* aspect in AspectJ

```
package exception;
public aspect AspectHandler {
 pointcut deliverMessage(Error error):
     handler(Error)&& args (error)
     && cflow(execution(* TestHandler.deliver(..)));
 before(Error error): deliverMessage(error){
     System.out.print(error);
     System.out.println("\n Exception Handled");}
}
```

As discussed in Section 2.3.1.4 , in AspectS, an exception handler block of *AsHandlerAdvice* is executed only when the sending of the message results in signaling an exception specified in the advice itself. Since SmallTalk does not have any special constructs for exception handling such as try or catch block of Java, a handler block of *AsHandlerAdvice* encompasses the existing block of code that handles the exception in the program. For example, as shown in Listing 4.8, if the value of a field *n* is not equal to zero, the method *dec* is used to decrement the value of *n*. However, method *dec* signals an exception and handles it whenever the value of *n* is zero. We assume that this program includes some other methods such as *inc* for incrementing the value of *n* as well.

Listing 4.8 Signaling exception in SmallTalk

```
dec
 |exception|
 (self n =0) ifTrue: [exception:=Error new.
       exception signal:'value of n cannot be less than 0'.
       self n: self n+10.
       Transcript show: 'Exception handled';cr.
       Transcript show: 'Value of n is set to '.
       Transcript show: self n;cr.
       ].
       self n: self n - 1.
       Transcript show: self n;cr.
```

Figure 4.2 shows the workspace code and transcript for the above example, in which the raised exception in handled by the *dec* method itself.

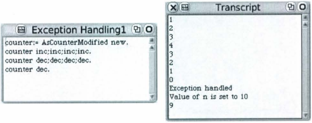


Figure 4.2: Workspace and output for exception handling example in SmallTalk

The above exception handling execution can be captured as a join point in AspectS. We assume that the *AspectHandler* aspect has a method named *adviceException* as shown in Listing 4.9. The *AsHandlerAdvice* object includes *Error* class as an exception. Whenever an error of type *Error* is signaled from the *dec* method, the exception is handled by the handler block.

Listing 4.9 Advising an exception handler join point in AspectS

```
adviceException
  ^AsHandlerAdvice
    qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific})
    pointcut: [{AsJoinPointDescriptor
        targetClass: AsCounterModified targetSelector: #dec}]
    exception: Error
    handlerBlock: [:receiver :arguments :aspect :client :ex | ex signal.
        Transcript show: 'Handler Block Is Executed'.].
```

Figure 4.3 shows the workspace code and transcript for the exception handling example with an *AsHandlerAdvice* associated with it. As we can see, the advice is triggered when an exception is raised within the *dec* method. Moreover, the handler block of the advice object replaces the existing handling block of *dec* method and handles the exception.

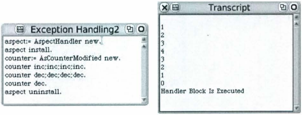


Figure 4.3: Workspace and output for exception handling example in AspectS

e) Class initialization join points

In AspectJ, a class initialization join point represents the loading of a class, including the initialization of the static part, e.g. class variables [1].

Listing 4.10 Class initialization in AspectS

```
Object subclass: #AsItem
  instanceVariableNames: 'id price'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'AspectS-ShoppingCart'
```

In Smalltalk a class initialization is done by passing arguments to the static method *subclass:instanceVariableNames:classVaiableNames:poolDictionaries:category:* of class Object or its subclasses as shown in Listing 4.10.

Since SmallTalk does not have any special syntax for class initialization and static methods cannot be advised in AspectS, class initialization join point selection cannot be implemented in AspectS.

f) Object initialization join points

In Java or C++, object initialization occurs when an object is created. AspectJ allows selecting the object initialization join point to perform certain additional object initialization [1]. However, in SmallTalk or AspectS, the object initialization is done by sending the message *new* to a class as shown below.

```
item1 := AsItem new.
```

Since *new* is a static method and cannot be selected as join point is AspectS, selection of object initialization join points is not permitted in AspectS.

g) Object pre-initialization join points

The object pre-initialization join point includes the passage from the constructor that was called first to the beginning of its parent constructor [1]. Since SmallTalk does not have any constructors, object pre-initialization join points cannot be selected in AspectS.

h) Advice execution join points

In AspectJ, the advice execution join point includes the execution of every advice in the system [1]. Using AspectS, execution of advice methods can be selected as join points. However, advices are objects and can be passed as method parameters in AspectS. Since AspectS does not provide any pointcut construct to select an object as join point, selecting the advice method execution as a join point will not imitate the advice execution join point of AspectJ.

4.1.2 Cross-cutting Concerns

Since the primary goal of introducing aspect-orientation was to modularize cross-cutting concern of a system or program into separate entity, each AOP language has the concept of cross-cutting concern. Hence, this feature is common for all AOP languages. For example, both in AspectJ and AspectS a cross-cutting concern contains aspects in the same way as packages (for AspectJ) or categories (for AspectS) contain classes.

4.1.3 Aspects

Since AspectML does not have any identical concept known as aspect, the aspect related features of AspectML are incomparable with the aspect related features of other two languages. Hence, we compare the aspects of AspectJ and that of AspectS in the following paragraphs.

a) Instantiation

As discussed in Section 2.3.1.1 , in AspectS, instances of the aspects are created by the user. Weaving occurs by sending an install message to an instance of the aspect. For example, using AspectS, an aspect *AsAbstractAspect* (Listing 4.9) can be instantiated like a regular class of SmallTalk as shown in Figure 4.4. If we create two instances of the same aspect and install those, the advice will be activated for each of the instances.

Listing 4.11 An aspect in AspectS

```
AsAspect subclass: #AsAbstractAspect
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'HelloWorld'
```

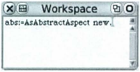


Figure 4.4: Aspect instantiation

On the other hand, in AspectJ by default, each aspect is a singleton, so one aspect instance is created automatically. Hence, unlike the aspects of AspectS, in AspectJ, the aspects cannot be directly instantiated [1].

b) Access specification

In AspectJ, like the classes or interfaces of Java, the visibility of the aspects can be specified by the access specifiers. Moreover, an aspect can have an access specifier of "privileged" in order to read and write the private members of the classes it is crosscutting [1]. Since SmallTalk does not allow access specification or visibility for the regular classes, the visibility cannot be specified for the aspects of AspectS too. This is not an issue in AspectS, since SmallTalk does not allow access specifiers.

c) Aspect precedence

Aspect precedence specifies the ordering of aspects and advices. Ordering of advices is important when advices of different aspects are applicable to the same join point in the system. [1] presents an example of aspect precedence, in which both the method *enter* and *exit* of *Home* class match the pointcuts of *HomeSecurityAspect* and *SaveEnergyAspect*. In order to see the advice execution in a desired order (as shown in Figure 4.5), it is necessary to set the precedence of aspects.

```
Engaging
Switching off lights
Exiting

Entering
Switching on lights
Disengaging
```

Figure 4.5 Output of aspect precedence example in AspectJ

In AspectJ, orders of advices are specified by declaring the precedence of aspects as shown below. In this example, *HomeSecurityAspect* will receive priority over *SaveEnergyAspect.*

```
declare precedence: HomeSecurityAspect, SaveEnergyAspect;
```

AspectS does not have any special construct to declare the precedence of the aspects. However, precedence to an aspect can be given by sending an install message to its instance prior to sending the install message to instances of other aspects. Since, in the workspace Figure 4.6, *HomeSecurityAspect* is installed prior to the installation of *SaveEnergyAspect*, it will get precedence over the second. Hence, we get the desired advice ordering as shown in Figure 4.5.

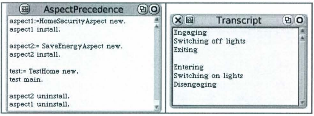


Figure 4.6: Workspace and output for aspect precedence example in AspectS

d) Nested Aspect

In AspectJ, an aspect may be defined either at the package level, or as a static member of a class, interface, or aspect. However, in AspectS, an aspect cannot be defined as a member of other classes or aspects; they can be only defined at the package level. For this reason unlike AspectJ, the aspects of AspectS cannot be embedded inside classes as nested aspects.

e) Implementing interfaces

In AspectJ, the aspects can implement interfaces. Since SmallTalk does not support interface, the aspects of AspectS cannot implement interfaces.

f) Abstract Aspects

Both in AspectJ and in AspectS, aspects can be abstract. An aspect containing an abstract method is an abstract aspect in AspectS. Since *methodAbstract* (Listing 4.12) belongs to the aspect *AsAbstractAspect* (Listing 4.11), *AsAbstractAspect* is an abstract aspect in AspectS.

Listing 4.12 Abstract method of AsAbstractAspect aspect

```
methodAbstract
    self subclassResponsibility
```

g) Extending classes and aspects:

In AspectJ, the aspects can extend classes and abstract aspects, but not concrete aspects [1, 16].

In AspectS, the aspects can inherit from both concrete and abstract aspects. For example, all the subclasses of *AsAspect* inherit *AsAspect*, which is a concrete aspect in AspectS.

Listing 4.13 Inheriting an abstract aspect in AspectS

```
AsAbstractAspect subclass: #SubAbstractAspect
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'HelloWorld'
```

Moreover, *SubAbstractAspect* (Listing 4.13) inherits an abstract aspect *AsAbstractAspect*. *SubAbstractAspect,* which is a subclass of the abstract aspect *AsAbstractAspect,* implements the abstract method *methodAbstract* (Listing 4.12) in Listing 4.14.

Listing 4.14 Method of SubAbstractAspect aspect

```
methodAbstract
    Transcript show:'Aspect can inherit
                     from abstract aspect'.
```

4.1.4 Pointcuts

a) Naming pointcut

In AspectML a pointcut is a set of functions and does not have a name [8]. However, both in AspectJ and AspectS, the pointcuts can be either *anonymous* or *named.* Since, in AspectS, pointcuts are objects too, they can be named by assigning them to variables. These named pointcuts can then be used later on in advices of aspects.

b) Types of pointcut

AspectJ allows a wide variety of pointcuts, which includes: kinded pointcuts, control-flow based pointcuts, lexical-structure based pointcuts, execution object pointcuts, argument pointcuts and conditional check pointcuts. Section 2.2.1.3 presented a general idea of the functions of each of those pointcut types.

*i. **Kinded pointcuts***

Kinded pointcuts have similar syntax to capture each kind of exposed join point such as in AspectJ. Section 4.1.1 provided a detailed overview of the exposed join point categories of AspectJ. In that section, we have already seen that some of those join points, such as: method execution join point, field access join point, and exception handler execution join point can be captured in AspectS too. We have also seen that, the join point selected by pointcuts along with the *receiverClassSpecific* advice qualifier attribute of AspectS is similar to the method execution join point of AspectJ. Although AspectS does not have any construct to select the method calls as join point, the join point selection by the *senderClassSpecific* advice qualifier attribute (Listing 4.15) of AspectS (if *NewTest* is added as a sender class to the aspect) can be emulated by combining the *call()* and *this*() pointcut of AspectJ as shown in Listing 4.16.

Listing 4.15 senderClassSpecific pointcut in AspectS

```
qualifier: (AsAdviceQualifier
    attributes: {#senderClassSpecific})
pointcut: [{AsJoinPointDescriptor
    targetClass: Test targetSelector: #deliver}]
```

Listing 4.16 Representation of senderClassSpecific pointcut in AspectJ

```
call (* Test.deliver(..))&& this(NewTest);
```

However, AspectJ does not have pointcut designators that can select specific instances. As a result, instance specific join point selection by the advice qualifier attributes receiverInstanceSpecific and *senderInstanceSpecific* of AspectS goes beyond the features available in JPM of AspectJ. Thus, these join point selection cannot be emulated using AspectJ.

ii. Control-flow based pointcuts

As discussed in the subsection b of Section 2.2.1.3 , in AspectJ, control-flow based pointcuts such as: cflow() and cFlowBelow() take another pointcut as argument. The advice qualifier attributes *cfFirstClass*, *cfAllButFirstClass*, *cfFirstInstance*, *cfAllButFirstInstance*, *cfFirstSuper* and *cfAllButFirstSuper* are used to specify the control-flow based pointcut in AspectS. Some of these pointcuts of AspectS can be emulated by combining the control-flow based pointcut of AspectJ along with some other simple pointcuts. For example, the join point selected by the pointcut and qualifier attribute *cfFirstClass* of AspectS in Listing 4.17 can be imitated by the combination of AspectJ pointcuts shown in Listing 4.18.

Listing 4.17 cfFirstClass pointcut in AspectS

```
qualifier: (AsAdviceQualifier
    attributes: { #receiverClassSpecific.#cfFirstClass. })
pointcut: [{ AsJoinPointDescriptor
    targetClass: AsFactorialM targetSelector: #factorial:. }]
```

Listing 4.18 Representation of cfFirstClass pointcut in AspectJ

```
!cflowbelow(execution(* AsFactorialM.factorial(..)))
     && execution(* AsFactorialM.factorial(..))
```

Similarly, the join point selected by the pointcut and qualifier attribute *cfAllButFirstClass* of AspectS in Listing 4.19 can be imitated by the combination of AspectJ pointcuts shown in Listing 4.20.

Listing 4.19 cfAllButFirstClass pointcut in AspectS

```
qualifier: (AsAdviceQualifier
     attributes: { #receiverClassSpecific.#cfAllButFirstClass. })
pointcut: [{ AsJoinPointDescriptor
     targetClass: AsFactorialM targetSelector: #factorial:. }]
```

Listing 4.20 Representation of cfAllButFirstClass pointcut in AspectJ

```
cflowbelow(execution(* AsFactorialM.factorial(..)))
     && execution(* AsFactorialM.factorial(..))
```

However, no cflow related advice qualifier attribute is used in either Listing 4.17 or Listing 4.19, the join points that are captured by the AspectS constructs can be emulated by the AspectJ pointcut shown in Listing 4.21.

Listing 4.21 Using cflow pointcut of AspectJ

```
cflow(execution(* AsFactorialM.factorial(..))) &&
execution(* AsFactorialM.factorial(..))
```

The join point selected by the pointcut and qualifier attribute *cfFirstSuper* of AspectS in Listing 4.22 can be imitated by the combination of AspectJ pointcuts shown in Listing 4.23.

Listing 4.22 cfFirstSuper pointcut in AspectS

```
qualifier: (AsAdviceQualifier
     attributes: { #receiverClassSpecific.#cfFirstSuper})
pointcut: [{ AsJoinPointDescriptor
     targetClass: FactorialM targetSelector: #factorialM:. }]
```

Listing 4.23 Representation of cfFirstSuper pointcut in AspectJ

```
!cflowbelow(execution(* FactorialM.factorial(..)))
    && execution(* FactorialM.factorial(..))
    && ! within(SubFactorialM)
```

Likewise, the join point selected by the pointcut and qualifier attribute *cfAllButFirstSuper* of AspectS in Listing 4.24 can be imitated by the combination of AspectJ pointcuts shown in Listing 4.25.

Listing 4.24 cfAllButFirstSuper pointcut in AspectS

```
qualifier: (AsAdviceQualifier
    attributes: { #receiverClassSpecific.#cfAllButFirstSuper})
pointcut: [{ AsJoinPointDescriptor
    targetClass: FactorialM targetSelector: #factorialM:. }]
```

Listing 4.25 Representation of cfAllButFirstSuper pointcut in AspectJ

```
cflowbelow(execution(* FactorialM.factorial(..)))
    && execution(* FactorialM.factorial(..))
    && !within(SubFactorialM)
```

However, as AspectJ does allow selecting specific instances, control-flow based instance specific pointcuts of AspectS such as *instanceFirst* and *instanceAllButFirst* cannot be emulated using AspectJ.

iii. Lexical-structure based pointcuts

Lexical-structure based pointcuts, such as *within()* and *withincode()* of AspectJ, capture join points occurring inside a segment of source code of specified classes, aspects and methods. AspectS does not allow such selection. However, as shown earlier in this section, to imitate some of the cflow-based pointcuts such as *cfFirstSuper* and *cfAllButFirstSuper* of AspectS, combining *within()* with control-flow based pointcuts of AspectJ is often necessary.

iv. Execution object pointcuts

In AspectJ, execution object pointcuts such as *this()* and *target()* pointcuts match the join points based on the types of the objects at execution time. The *this()* pointcut selects all the join points associated with the current object, whereas the *target()* pointcut is used to select the join points associated with the object on which the method is invoked. AspectS has the limitation to select join points based on the types of objects at execution time. However, when using AspectJ, to imitate some join point selection of AspectS, we might need to combine a *this()* or a *target()* pointcut along with some other pointcuts. For example, in Listing 4.16 we have seen how a join point selection by a *senderClassSpecific* advice qualifier attribute of AspectS is emulated by combining a *call()* and *this()* pointcut of AspectJ.

v. Argument pointcuts

The *args()* pointcuts can expose the context at the matched join point in AspectJ. AspectS passes execution context automatically as arguments into the advice. For example, in AspectS, in order to see the arguments passed in each method execution of a program related to control-flow based pointcut, we can simply use a regular before advice as shown in Listing 4.26.

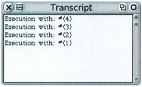


Figure 4.7: Printing arguments in AspectS

Listing 4.26 Printing argument in AspectS

```
adviceClassAllButFirst
     ^ AsBeforeAfterAdvice
     qualifier: (AsAdviceQualifier
       attributes: { #receiverClassSpecific.#cfAllButFirstClass. })
     pointcut: [{ AsJoinPointDescriptor
       targetClass: AsFactorialM targetSelector: #factorial:. }]
     beforeBlock: [:receiver :arguments :aspect :client |
       Transcript show: 'Execution with: '.
       Transcript show: arguments;printString; cr.]
```

However, to see the same output in AspectJ, we need to add an additional *args()* pointcut (Listing 4.27) along with the existing pointcut combination.

Listing 4.27 Printing argument in AspectJ

```
public aspect AspectClassAllButFirst {
  before(int arg):cflowbelow(execution(* AsFactorialM.factorial(..)))
     && execution(* AsFactorialM.factorial(..))
     && args(arg) {
     System.out.println("Execution with: "+arg);
     }
}
```

vi. Conditional check pointcuts

In AspectJ, the conditional check pointcut captures join points based on some conditions. The conditions are to be checked at the join point. AspectS and AspectML do not allow a join point to be selected based on such checking. Thus, conditional check pointcuts cannot be imitated in AspectS and AspectML.

c) Pointcut operators

AspectJ provides a unary negation operator (!) and two binary operators (|| and &&) to form more complex matching rules [1]. Where the negation (!) allows the matching of all join point except those specified by the pointcut, the binary operators (|| and &&) are used to combine pointcuts. Combining two pointcuts with the || operator causes the selection of join points that match either of the pointcuts, whereas combining them with the && operator causes the selection of join points matching both the pointcuts. In AspectS, the method *difference:* that takes an *AsJoinPointDescriptor* object as argument can be used to emulate AspectJ's negation (!) operator. However, the negation operator of AspectS can be used with a single pointcut, whereas method *difference:* needs at least two join point objects as shown in Table 4.1 (Third row). AspectS also allows combining pointcuts by using the methods related to set operations such as: *union:* and *intersection:*. Combining two pointcuts with the method *union:* causes the selection of join points that match either of the pointcuts, whereas combining them with the method *intersection:* causes the selection of join points matching both the pointcuts. Listing 4.28 represents two pointcut objects *jpset1* and *jpset2*. *jpset1* is a set of two join points : the first selects execution of the method *addItem:* of *AsInventory* class and the second selects execution of the method *removeItem:* of the same class. Similarly, *jpset2* is also a set of two join points: the first selects execution of the method *addItem:* of *AsInventory* class and the second selects execution of the method *addItem:* of *AsShoppingCart* class.

Listing 4.28 Pointcut objects in AspectS

```
jpset1:={AsJoinPointDescriptor
       targetClass: AsInventory targetSelector: #addItem:.
          AsJoinPointDescriptor
       targetClass: AsInventory targetSelector: #removeItem:.}.
jpset2:={AsJoinPointDescriptor
       targetClass: AsInventory targetSelector: #addItem:.
     AsJoinPointDescriptor
       targetClass: AsShoppingCart targetSelector: #addItem:}.
```

Each of the above pointcut objects can be represented using the pointcut designators of AspectJ (Listing 4.29). A set of two *AsJoinPointDescriptor* objects of AspectS is similar to an AspectJ pointcut that combines two join point with an || operator.

Listing 4.29 Emulating Listing 4.28 in AspectJ

```
pointcut jpset1() : execution (*
AsInventory.addItem(..)) || execution (*
AsInventory.removeItem(..));
pointcut jpset2() : execution (*
AsInventory.addItem(..)) || execution (*
AsShoppingCart.removeItem(..));
```

Based on the above sets of join points of AspectS (Listing 4.28) and pointcuts of AspectJ (Listing 4.29), the following table (Table 4.1) shows how join points, those are selected using the methods related to set operations of AspectS, can be emulated using the pointcut operators of AspectJ:

Table 4.1: Pointcut operators in AspectJ and AspectS

AspectS	AspectJ
`jpset3:=jpset1 union: jpset2.`	`pointcut jpset3() :jpset1() || jpset2();`
`jpset4:=jpset1 intersection: jpset2.`	`pointcut jpset4() :jpset1() && jpset2();`
`jpset5:=jpset1 difference: jpset4.`	`pointcut jpset5() :jpset1()&&!jpset4();`

AspectML does not have any pointcut designator to combine join points as shown above.

d) First class pointcut

A first class pointcut is a pointcut that can be passed as a method parameter or can be assigned to a variable, i.e. is an instance of the primary modularization mechanism, in this case an object, instance of a class. AspectML allows passing pointcuts as method parameters. Listing 3.6 of Chapter 3 shows how in AspetML a pointcut can be passed as a method argument.

In AspectS, since pointcuts are objects, besides passing them as method parameters, it is also possible to assign the pointcut objects to the variables. The pointcut objects *jpset1* and *jpset2*, created in Listing 4.28, are first-class pointcuts. As we have seen in the previous section, these pointcuts were passed as a method parameter to the methods *union:*, *intersection:* and *difference:* of AspectS.

AspectJ does not have the construct for first class pointcut. As a result, pointcuts cannot be passed as arguments in AspectJ.

4.1.5 Advice

Advices of AspectJ can be considered as the methods of Java. However, they have some differences with the regular methods such as:

- they do not have a name,
- they cannot be called directly (it is the system's job to execute them) ,
- they do not have access specifiers.

In AspectS, the advices are objects of the class *AsAdvice*. If assigned to a variable, the advice objects in AspectS can have names. However, they do not have an access specifier. Advices in AspectS are enabled or disabled by sending an install or uninstall message to the class *AsAspect* or to the instance of its subclass. Furthermore, an aspect cannot de-activate itself as part of an advice block. However, as shown in Listing 4.30, an aspect can install and uninstall other aspects as part of its advice block.

Listing 4.30 Installing and uninstalling other aspect as part of an advice block

```
adviceBefore
|after|
    ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier
                attributes: {#receiverClassSpecific})
    pointcut: [{AsJoinPointDescriptor
               targetClass: Test targetSelector: #deliver}]
    beforeBlock: [:receiver :arguments :aspect :client |
           Transcript show: '  "Hello World"   '.
           after :=AfterHelloWorld new.
           after install.
           after uninstall.]
```

The above advice belongs to the *BeforeHelloWorld* Aspect. It selects the execution of deliver method, that prints the message "Wanna learn AspectS?". We only install and uninstall *BeforeHelloWorld* in the workspace (Figure 4.8).

Figure 4.8: Installing and uninstalling *BeforeHelloWorld* Aspect

However, as part of its advice block it installs and uninstalls *AfterHelloWorld* Aspect, which contains an advice that prints the message "Goodbye World" (Figure 4.9) after execution of the method *deliver*.

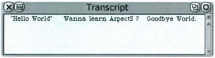


Figure 4.9: Result of installing and uninstalling *AfterHelloWorld* aspect

An Advice in AspectML is composed of two parts, the body and the pointcut designator. Like the advices of AspectJ, in AspectML an advice possess following characteristics:

- It does not have a name ,
- It cannot be called directly (it is the system's job to execute it) ,
- It does not have access specifiers.

AspectJ supports three types of advices before advice, after advice and around advice. A before advice allows adding some new behavior before any particular join point. This acts similar to the *AsBeforeAfterAdvice* with a before block in AspectS. The after advice of AspectJ, which allows placing some additional behavior after a join point, can be imitated in AspectS by an *AsBeforeAfterAdvice* with an after block in it. In AspectJ when both before advice and after advice is used in the same aspect, advice acts similar to the AsBeforeAfterAdvice with both the before and after code block of AspectS. An around advice of AspectJ, that adds new behavior or modifies some existing behavior of the program around a join point, is similar to the *AsAroundAdvice* of AspectS.

AspectS supports two other types of advices based on the classes: *AsIntroductionAdvice* and *AsHandlerAdvice*. With *AsIntroductionAdvice* one can introduce new behavior that is needed in the aspect's context. The operation of *AsIntroductionAdvice* can be emulated by the static crosscutting feature *Introduction* of AspectJ. An AsHandlerAdvice that selects an exception handler execution join point in AspectS can be emulated by the exception pointcut of AspectJ.

AspectML does not provide any special keyword to distinguish between its advices. As a result all the advices in AspectML look similar. However, application of advices varies with the *trigger time* (before, after, or around) of pointcut designators [8].

4.1.6 Static Crosscutting

Static crosscutting features such as the *Introduction* of AspectJ allows for introducing new behavior, which is needed in aspect's context. Although *AsIntroductionAdvice* is placed under the category of advices in AspectS, this advice acts similarly to the *Introduction* of AspectJ. The following bulleted lists present a detailed overview of the functions allowed to be performed using the static crosscutting feature of each of our three experimental languages:

AspectJ:

- The *introduction* is a static crosscutting instruction that introduces changes to the classes, interfaces, and aspects of the system. For example, introductions can add a method or field to a class.
- Type-hierarchy modification is a static crosscutting instruction that allows modifying the inheritance hierarchy of existing classes to declare a superclass and interfaces of an existing class without breaking the rules of Java language [1].
- The *compile-time declaration* is a static crosscutting instruction that allows the adding of compile-time warnings and errors upon detecting certain usage patterns.

AspectS:

- With an introduction advice (*AsIntroduction*) one can introduce new behavior that is needed in the aspect's context. The added behavior may be invoked by the aspect, and may actively invoke the aspect's or client's behavior itself
- Introductions into method wrappers is not allowed [6].
- A method that is understood but not implemented by a class can be introduced [6].
- Type-hierarchy modification is not possible in AspectS.
- In AspectS weaving happens during runtime. Thus, adding compile-time warnings and errors upon detecting certain usage patterns is not possible in AspectS.

AspectML:

AspectML does not support static crosscutting features such as introduction, type-hierarchy modification or compile-time declaration.

4.2 Discussion

AspectJ exposes several categories of join point. The exposed join point categories of AspectS and AspectML are still very limited compared to that of AspectJ. The following table (Table 4.2) provides the summarized version of our discussion on exposed join point of three languages (Section 4.1.1).

Table 4.2: Exposed join point categories of AspectJ, AspectS and AspectML

	AspectJ	AspectS	AspectML
Method Execution	√	√	×
Method Call	√	×	√
Constructor Execution	√	×	×
Constructor Call	√	×	×
Field Read Access	√	×	×
Field Write Access	√	×	×
Exception handler execution	√	×	×
Class initialization	√	×	×
Object initialization	√	×	×
Object pre-initialization	√	×	×
Advice execution	√	×	×

Although AspectS does not have any pointcut construct to select the join point related to field access or exception handler execution, those join points can be emulated indirectly using some other features available in AspectS. For example, in AspectS, an advice (*AsHandlerAdvice)* is used to advice the exception handler execution join point. The function of an exception handler execution pointcut of AspectJ can be emulated using this advice of AspectS.

AspectJ provides pointcut designators to select a number of different join points. On the other hand, not having very rich pointcut constructs like AspectJ, AspectS and AspectML do not allow selecting join point based on the following pointcuts shown in Table 4.3.

Table 4.3: Pointcuts those are not available in AspectS and AspectML

Some pointcuts of AspectJ	
initialization()	within()
preinitialization()	withincode()
Staticinitialization()	if()

AspectS allows combining join points using the methods related to set operations. This imitates the functionality of pointcut operators of AspectJ.

Except for the instance specific attributes, AspectJ allows emulating the point cuts specified by the other advice qualifier attributes of AspectS as shown in Table 4.4.

Table 4.4: AspectS Pointcuts that can or cannot be emulated using AspectJ

AdviceQualifier Attributes in AspectS		**Can or cannot be emulated in AspectJ**
Receiver or sender aware activation	cflow activation	
receiverClassSpecific	-	√
senderClassSpecific	-	√
receiverInstanceSpecific	-	×
senderInstanceSpecific	-	×
receiverClassSpecific	Class First	√
receiverClassSpecific	Class All-But-First	√
receiverClassSpecific	Instance First	×
receiverClassSpecific	Instance All-But-First	×
receiverClassSpecific	Super First	√
receiverClassSpecific	Super All-But-First	√

Based on the previous table, Table 4.5 presents a mapping between the pointcuts of AspectS and AspectJ.

Table 4.5: Imitating the AspectS join point selection using AspectJ

AspectS	**AspectJ**
`qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific}) pointcut: [{AsJoinPointDescriptor targetClass: Test targetSelector: #deliver}]`	`execution (* Test.deliver(..));`

qualifier: (AsAdviceQualifier attributes: {#senderClassSpecific}) pointcut: [{AsJoinPointDescriptor targetClass: Test targetSelector: #deliver}] Note: NewTest should be added as a Sender Class	call (* Test.deliver(..))&& this(NewTest);
qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific}) pointcut: [{AsJoinPointDescriptor targetClass: AsFactorialM targetSelector: #factorial:.}]	cflow(execution(* AsFactorialM.factorial(..))) && execution(* AsFactorialM.factorial(..))
qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfFirstClass.}) pointcut: [{AsJoinPointDescriptor targetClass: AsFactorialM targetSelector: #factorial:.}]	!cflowbelow(execution(* AsFactorialM.factorial(..))) && execution(* AsFactorialM.factorial(..))
qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfAllButFirstClass.}) pointcut: [{AsJoinPointDescriptor targetClass: AsFactorialM targetSelector: #factorial:.}]	cflowbelow(execution(* AsFactorialM.factorial(..))) && execution(* AsFactorialM.factorial(..))
qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfFirstSuper}) pointcut: [{AsJoinPointDescriptor targetClass: FactorialM targetSelector: #factorialM:.}]	!cflowbelow(execution(* FactorialM.factorial(..))) && execution(* FactorialM.factorial(..)) && ! within(SubFactorialM)
qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfAllButFirstSuper}) pointcut: [{AsJoinPointDescriptor targetClass: FactorialM targetSelector: #factorialM:.}]	cflowbelow(execution(* FactorialM.factorial(..))) && execution(* FactorialM.factorial(..)) && !within(SubFactorialM)

All three of our experimental languages have constructs to advice before, after and around any join points.

The functions of *AsIntroductionAdvice* and *AsHandlerAdvice* of AspectS can be imitated by the static crosscutting feature *Introduction* and by advising an exception handler execution join point respectively.

The join point model of AspectJ is much richer than that of either AspectS or AspectML. However, the latter two languages provide dynamic AOP capabilities, which is not available in AspectJ.

Chapter 5

ASPECT-ORIENTED MODELING IN UML

5.1 Related Works

While aspect-oriented programming (AOP) is rapidly maturing, there is still not enough support from the commercial modeling tools for aspect-orientation at software modeling level. Although many modeling tools are based on UML [28], it lacks specific constructs for aspects and their associated concepts [12]. However, the standardized extension mechanisms offered by UML can be used to provide aspect-oriented modeling facilities. This extension mechanism of UML is known as profile. Profiles allow adding user-defined categories of UML model elements by referring to a base class, which is a class in the UML meta-model such as Class and Association. Profiles are defined using stereotypes, tag definitions, and constraints. A stereotype defines how an existing metaclass (or other stereotype) may be extended. Certain stereotypes are predefined in the UML; others are usually defined by users. Stereotypes are also used to specify additional constraints and tag definitions. Tagged definitions allow specifying user-defined meta-attributes for a model element. Constraints allow specifying semantics or usage for a model element. Both tag definitions and constraints should be defined in conjunction with a stereotype.

An overview of some of the prior works for modeling aspects in UML is presented in [29]. The early work is based on the extension mechanisms in UML 1.x versions. Since these mechanisms are not fully integrated with the meta-model, the specification of advices and pointcuts often remains in textual form [30, 31, 32] and requires special model parsers for code generation.

[33], which is a later extension to [32], presented aspects as stereotyped classes. However, it was not a meta-model based profile. Rather than providing an aspect extension, the connection between aspects and base-model is made as part of the model.

Initial work presented in [34] proposed the specification of aspects as stereotypes on classes and was later extended to include advice and pointcut specification [35]. It models cross-cutting associations to show which aspect features relate to which base model elements. Thus, it gives a clear separation of aspects and base system, which is the primary objective of AOSD.

[36] proposes a profile for AspectJ. This profile represents messages in collaborations as join points, advices and pointcuts as stereotyped operations, and introduction of fields or methods as templated collaborations. Also, in this profile, the connection to the base features is made via dependencies in the model [12].

An earlier proposal for aspect modeling using UML 2.0 was presented in [37], however without fully defining an extension profile.

Other existing works are based on defining new UML meta-classes instead of defining stereotypes for existing meta-classes. This approach requires specialized tools to support the introduced meta-classes [38, 39]

One of the prior works on aspect modeling in UML proposes join point annotations for UML [40]. [41] describes a translation of aspect UML to object-oriented Petri-nets. However, this translation is limited to pointcuts around method calls.

Using the standard UML extension mechanisms, [42] provides suitable representations for all components of an aspect (such as join points, pointcuts, pieces of advice, and introductions) as well as for the aspect, itself. The representations are supplied with supplementary meta-attributes to hold the weaving instructions. Furthermore, the approach implements AspectJ's weaving mechanism in the UML and specifies a new relationship signifying the crosscutting effects of aspects on their base classes. However, as [42] is not based on UML profile, it requires special tool support.

Using the extension mechanisms in UML 2.0, [12] presents a meta-model, which is a UML profile for AspectJ language (Figure 5.1). It also offers a translation to code. The approach followed in it offers the following advantages over previous proposals:

- The extension requires no special software support and allows aspect modeling to be used within existing, mature software tools. This contrasts with earlier proposals [38, 39], which cannot be used with available modeling tools and require specific tool support.
- The proposed technique is supported by UML XMI model interchange facilities. The model extension, as well as any models it is applied to, can be exchanged between different MOF (Meta-Object-Facility) compliant UML modeling tools.

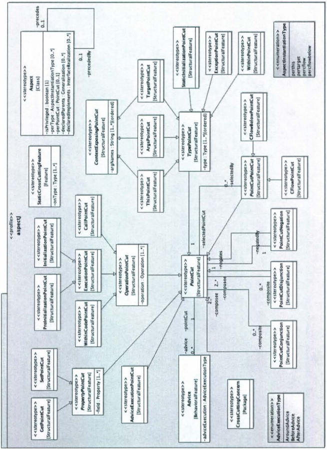


Figure 5.1: AspectJ Profile

- It allows all aspect-related concepts to be specified in meta-model terms. The models can be easily manipulated or verified without requiring the parsing of keywords or other textual specifications by special tools.
- It maintains strict separation of base-model and cross-cutting concerns.

However, this profile is not a generic aspect-oriented modeling extension and cannot be used for specification of a platform independent model (PIM). Moreover, the profile allows the specification of a platform-specific model (PSM), namely one that is specific to the Java and AspectJ platform. Since AspectJ follows static AOP approach, the extension does not support dynamic AOSD.

[43] presents a UML 2 profile for platform-independent modeling (PIM) with advanced pointcut expressions and a corresponding model weaving mechanism for behavior models using UML 2 Actions.

Recent work on Aspect-Oriented Frameworks (AOF)-based development is presented in [44]. It proposes UML-AOF, an UML profile for modeling a kind of AOF (termed as CF in the proposal) which encapsulates just one crosscutting concern. The proposed profile uses the Evermann's profile [12] as base AOP. However, like [12], presented profile does not support the specification of platform-independent model.

In summary, much of the existing work on AOM profiles for UML is either based on older UML versions, not well integrated on the meta-model level. However, based on AspectJ, [12] presents a complete UML profile, which is well integrated on the meta-model level. It also does not require any specific tool support. Inspired by this work, in this research we propose to extend this profile for other AOP languages. The proposed profile will allow the specification of platform-independent model by providing the modeling facility for both static and dynamic AOSD.

5.2 Our Approach

AOM approaches can be distinguished along two orthogonal dimensions: the level of weaving and the symmetry of the approach. Our work is positioned at the asymmetric code-weaving level. The aspect-oriented model is converted to aspect-oriented code, which can be woven by an aspect-oriented compiler. We also make a clear distinction between the base-system and the cross-cutting concerns (Figure 5.2).

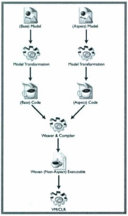


Figure 5.2: Our AOP Approach in Context (adapted from [12])

We present our UML meta-model for a selection of core aspect-oriented constructs. Rather than specializing UML meta-classes, we extend them using UML stereotypes.

As a result, the developed model becomes a meta-model, which is a profile and can be applied to other UML models.

The previously developed UML extension for static AOP treats aspects as extensions of the Class meta-class, i.e. a stereotyped class. Within that framework, pointcuts are stereotyped structural features and advices are stereotyped behavioral features, typically operations.

However, this approach is not feasible for dynamic AOM, because dynamic approaches represent AOSD concepts as first-class modules. For example, join point descriptors (pointcuts), advice and aspects are all objects in AspectS, while pointcuts are functions in AspectML. Thus, our approach will differ from the existing work in [12] by providing appropriate extensions.

5.3 Modeling Elements

This section presents modeling elements for core generic meta-model of the AOP languages. The elements to be modeled are selected based on the core generic features of AOP languages

5.3.1 CrossCuttingConcern

Both AspectJ and AspectS have the concept of cross-cutting concern that acts like a package and contains aspects of the language. Thus, a modeling element CrossCuttingConcern (Figure 5.3) is introduced as a way of grouping related aspects of AOP languages in the modeling level. We define a stereotype CrossCuttingConcern that extends UML meta-class Package. In any UML model, a package stereotyped as «CrossCuttingConcern» will represent a crosscutting concern for that model. Since the UML meta-model already specifies that packages own classes, the CrossCuttingConcern meta-class does not need to be associated with the Aspect meta-class.

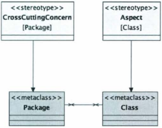


Figure 5.3: Cross cutting concern as package extension (adapted from [12])

5.3.2 Aspect

Recall Section 4.1.3, where we compared the aspect of AspectJ and AspectS. In both languages, aspects can have instance variables, class variables, instance methods, and class methods. The behavior of an aspect is similar to that of a class. Thus, an Aspect (Figure 5.4) can be modelled as a stereotype that extends the existing UML meta-class **Class**. In a model, within a package stereotyped as «CrossCuttingConcern», any class stereotyped as «Aspect» will represent the aspect for the model.

Since this proposal is positioned in asymmetric AOM (Figure 5.2), elements of the cross-cutting concern model or models must remain separated from base-model elements. The following constraint ensures this by requiring that classes that are stereotyped as «Aspect» are only packaged in packages that are stereotyped as «CrossCuttingConcern».

context Aspect inv:

package.oclIsKindOf(CrossCuttingConcern)

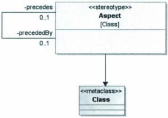


Figure 5.4: Aspect as a class extension in new profile

In this profile Aspect precedence is modelled as a recursive relationship between aspects. Each aspect has at most one directly preceding and following aspect.

The AspectJ and AspectS specification state that aspects may extend classes or other aspects but that classes may not extend aspects. Consequently, we add the following constraint that ensures for all generalizations that the specific class of a general class that is an aspect is also an aspect:

context Generalization inv:

general.oclIsKindOf(Aspect) implies specific.oclIsKindOf(Aspect)

5.3.3 Advice

In AspectJ, an advice is similar to a regular method. An advice of AspectJ can be modeled as Behavioral Feature. However, in AspectS advices are objects, which cannot be modeled by extending the meta-class **BehavioralFeature**. Also, if advices are modeled extending the meta-class BehavioralFeature, advices could not be passed as method arguments or be assigned to variables. To facilitate dynamic modeling an advice should be modeled as object. In the profile we model an advice using the meta-class Advice (Figure 5.5), which extends the UML meta-class Class.

Since advices are used by aspects, the meta-class Advice is associated with the meta-class Aspect.

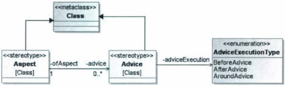


Figure 5.5: Advice as a class extension in new profile

Both in AspectJ and AspectS, advice code can be executed before, after, or around a pointcut. We model adviceExecution as an attribute of the Advice meta-class. The values are provided by the enumeration AdviceExecutionType. The meta-class Advice with its attribute adviceExecution will allow modeling before, after, and around advices of both AspectJ and AspectS.

5.3.4 Joinpoint

In AspectJ, a pointcut is used to select a join point. A pointcut can select either a single join point or a combination of one or more join points. On the other hand, in AspectS, a pointcut is a set of join points (Section 2.3.1.3), where each join point is described by an object of AsJoinPointDescriptor class. We consider that each advice is associated with a pointcut that is a set of join points.

In the profile, a join point is modeled using the meta-class Joinpoint (Figure 5.6) and considered as a set (pointcut) consists of a single join point. Since in dynamic AOP join points are objects, in the new profile the meta-class Joinpoint extends the UML meta-class Class.

Joinpoint is an abstract meta-class. Rather than specifying the type and textual declaration of join points as attributes on Joinpoint, we subclass the Joinpoint meta-class to allow different attributes to be modelled for different join points.

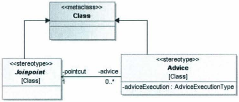


Figure 5.6: Joinpoint as a class extension in new profile

Because pointcuts are used by advices, the meta-class Joinpoint is associated with the meta-class Advice.

5.3.5 Join Point Composition

Earlier in Section 4.1.4c) and Table 4.1, we showed how pointcut operators of AspectJ can be emulated using methods related to set operations of AspectS. Since the composed join point is common between these two languages, we introduce three meta-classes to model join point composition (Figure 5.7): JoinpointConjunction, JoinpointDisjunction, and JoinpointNegation. The stereotype JoinpointConjunction will allow to model the composition of at least two join points that are composed with an && operator of AspectJ. It will also allow modeling the composition of at least two AsJoinPointDescriptor objects that are composed using the method *intersection:* of AspectS. Similarly, the stereotype JoinpointDisjunction will allow modeling the composition of at least two join points that are composed with a || operator of AspectJ. It will allow modeling the composition of at least two AsJoinPointDescriptor objects that are composed using the method *union:* of AspectS.

A modeler must make sure that the number and type of arguments are consistent for all the join points that are part of a join point disjunction or join point conjunction. We add the following constraints:

context JoinpointConjunction inv:

self.hasParts→forAll(P1,P2:ExecutionJoinpoint|oclITypeOf(P1.operation.ownedparameter)= oclITypeOf(P2.operation.ownedparameter))

context JoinpointDisjunction inv:

self.hasParts→forAll(P1,P2:ExecutionJoinpoint|oclITypeOf(P1.operation.ownedparameter)= oclITypeOf(P2.operation.ownedparameter))

Since the negation operation accepts only a single operand, while conjunction and disjunction require at least two, we model these join point compositions as separate sub-classes. Ordering of the operands for conjunction or disjunction is not necessary, since the operations are associative and commutative.

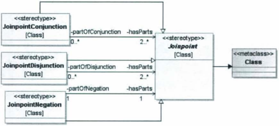


Figure 5.7: Joinpoint compositions in the profile

5.3.6 ExecutionJoinpoint

In Section 4.1.1a) , we showed how receiverClassSpecific advice qualifier attribute can be used to emulate the function of an execution pointcut of AspectJ. As a result, we can consider that both AspectJ and AspectS have pointcut designator to select method execution join point. On the other hand, as AspectS does not provide any construct to select call join point, a call pointcut is not common to the two languages.

In our profile, we model the selection of execution join point using the meta-class ExecutionJoinpoint. The single valued attribute operation allows selecting the method whose execution will be selected as join point. This will allow modeling the following code snippets from AspectJ and AspectS respectively:

```
pointcut deliverMessage()
     : execution (* Test.deliver(..));
```

```
qualifier: (AsAdviceQualifier
     attributes: {#receiverClassSpecific})
pointcut: [{AsJoinPointDescriptor
     targetClass: Test targetSelector: #deliver}]
```

However, AspectS does not allow advising the execution of a static method as a join point. We add the constraint that only non-static methods can be selected as the values of the tag operation of classes stereotyped as «ExecutionJoinpoint»:

context ExecutionJoinpoint inv:

self.operation.isStatic=False

Although there is no call join point in AspectS, we can emulate one by specifying the senderClass of a senderclassSpecific join point. If the sender class is known, we can find the source for this method execution, which is actually the call to that method. On the other hand, the call() and this() pointcuts (Listing 4.16) in AspectJ can be used to emulate a senderClassSpecific join point in AspectS.

A call join point can be modeled using a separate meta-class such as CallJoinpoint. Since not all AOP languages have the call pointcut, there is no meta-class for this in the profile. Moreover, as we can translate an execution to a call, it is unnecessary to model a separate call join point. For this reason, as shown in

Figure 5.8, rather than keeping a separate meta-class CallJoinpoint, a single valued attribute senderClass of ExecutionJoinpoint meta-class is included to specify at most one class as sender.

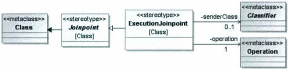


Figure 5.8: ExecutionJoinpoint in new profile

The data type of senderClass could be Classifier, since AspectJ allows method call from an object of a class that realizes an interface. However, AspectS does not have the concept of interface. As a result, to restrict modeling the method calls that are only from classes, the data type of senderClass is set to Class.

The ExecutionJoinpoint meta-class will allow modeling the code snippets shown in Listing 4.15 (AspectS code) and Listing 4.16 (AspectJ code).

5.3.7 ExceptionJoinpoint

In AspectS, an AsHandlerAdvice is used to advise an exception handler execution join point. For this reason, an AsHandlerAdvice could be modeled as one of the values of the enumeration AdviceExecutionType shown in Figure 5.5. However, as shown in chapter 4 (Section 4.1.1d)) the operation of AsHandlerAdvice of AspectS is similar to the exception pointcut of AspectJ. Instead of modeling the handler advice kind of AspectS, we model selection of exception join points using the meta-class ExceptionJoinpoint (Figure 5.9). The attribute exceptionClass of type Class is used to specify an exception class for the join point.

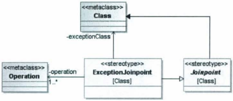


Figure 5.9: ExceptionJoinpoint as a class extension

The meta-class ExceptionJoinpoint will allow modeling the exception join point specified by the following code snippets of AspectS.

```
qualifier: (AsAdviceQualifier
      attributes: {#receiverClassSpecific})
   pointcut: [{AsJoinPointDescriptor
      targetClass: TestHandler targetSelector: #deliver}]
   exception: Error
   handlerBlock: [:receiver :arguments :aspect :client :ex |
      ex signal.].
```

If the above code of AspectS is modeled using the profile, this will be an equivalent model for an AspectJ join point as shown below.

```
pointcut deliverMessage(Error error): handler(Error)
         && args (error)
         && cflow(execution(* TestHandler.deliver(..)));
```

However, due to compiler limitations only before advice is supported by AspectJ to advise a handler join point. As a result, we add the constraint that an exception join point can only be advised by a before advice:

context ExceptionJoinpoint inv:

self.ofPointcut.ofAdvice→forAll(a:Advice|a.adviceExecution=BeforeAdvice)

5.3.8 PropertyJoinpoint

AspectS does not have any pointcut constructs to select the read or write access to the fields as join points. However, in chapter 4 (Section 4.1.1c)), we have already seen how the field read or write access join points of AspectJ can be emulated using AspectS. Therefore, we include these join points in the profile.

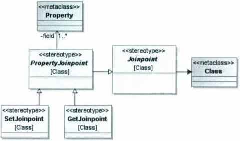


Figure 5.10: PropertyJoinpoint as a class extension

PropertyJoinpoint (Figure 5.10) is a superclass of those types of join points that are associated with reading and writing fields. It possesses a multi-valued attribute field with data type Property. This will allow modeler to select a field (from the base model), whose access will be selected as a join point. From example, if a modeler wants to model the *get()* pointcut shown in Listing 5.1, he needs to create a class, say getN, and apply Stereotype GetJoinpoint to it; the class getN will be stereotyped as «GetJoinpoint». As a result, it will have a tag definition *field.* The field *n* of the class *AsCounterModified1* should be selected form the base model as the value of the tag field. Similarly, the *set()* pointcut from the same listing can be modeled using the modeling element SetJoinpoint of the profile.

Listing 5.1 Field access pointcuts of AspectJ

```
pointcut getN(): get(private int AsCounterModified1.n);
pointcut setN(): set(private int AsCounterModified1.n);
```

5.3.9 CFlowJoinpoint

In Table 4.4, we showed some of the cflow related pointcuts of AspectS that can or can not be emulated using AspectJ constructs. Table 4.5 indicated that the join point selections by the cflow related advice qualifier attributes such as cfFirstClass, cfAllButFirstClass, cfFirstSuper, and cfAllButFirstSuper have equivalent pointcuts in AspectJ. The advice qualifier attributes related to cflow in AspectS can be emulated by using the combination of cflowbelow pointcut, execution pointcut and args pointcut of AspectJ (from Listing 4.17 to Listing 4.20 and from Listing 4.22 to Listing 4.25). On the other hand, using the combination of cflow pointcut, execution pointcut and args pointcut of AspectJ it is possible to emulate those pointcuts in AspectS which select the execution of a recursive method without using any qualifier attributes related to cflow (Listing 4.21). Based on the above emulation, we decide to introduce a modeling element CFlowJoinpoint (Figure 5.11) that will allow modeling the cflow based pointcuts that can be translated back and forth between AspectJ and AspectS.

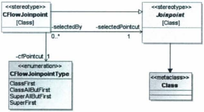


Figure 5.11: CFlowJoinpoint as a class extension

The join points selected by the ClassFirst, ClassAllButFirst, SuperFirst and SuperAllButFirst advice qualifier attributes of AspectS can be selected using the pointcut constructs of AspectJ. For this reason we consider those join points as common between these languages. We model cfPointcut as an attribute of the CFlowJoinpoint meta-class. The values are provided by the enumeration CFlowJoinpointType. When the profile is applied to a model, cfPointcut becomes a tag of stereotyped «AdviceCollection». Its values ClassFirst, ClassAllButFirst, SuperFirst, and SuperAllButFirst respectively represent the advice qualifier attributes: cfFirstClass, cfAllButFirstClass, cfFirstSuper, and cfAllButFirstSuper of AspectS.

Using the meta-class CFlowJoinpoint, pointcuts shown in Table 5.1 can be modeled using the profile.

Table 5.1: Pointcuts that can be modeled using the meta-class CFlowJoinpoint

AspectS	AspectJ
`qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfFirstClass.})` `pointcut: [{AsJoinPointDescriptor targetClass: AsFactorialM targetSelector: #factorial:.}]`	`!cflowbelow(execution(* AsFactorialM.factorial(..))) && execution(* AsFactorialM.factorial(..))`
`qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfAllButFirstClass.})` `pointcut: [{AsJoinPointDescriptor targetClass: AsFactorialM targetSelector: #factorial:.}]`	`cflowbelow(execution(* AsFactorialM.factorial(..))) && execution(* AsFactorialM.factorial(..))`
`qualifier: (AsAdviceQualifier attributes: {#receiverClassSpecific.#cfFirstSuper})` `pointcut: [{AsJoinPointDescriptor targetClass: FactorialM targetSelector: #factorialM:.}]`	`!cflowbelow(execution(* FactorialM.factorial(..))) && execution(* FactorialM.factorial(..)) && ! within(SubFactorialM)`
`qualifier: (AsAdviceQualifier`	`cflowbelow(execution(*`

attributes: {**#receiverClassSpecific.#cfAllButFirst Super**} pointcut: [{AsJoinPointDescriptor targetClass: FactorialM targetSelector: **#factorialM:**.}]	FactorialM.factorial(..))) && execution(* FactorialM.factorial(..)) && !within(SubFactorialM)

The meta-class Joinpoint is associated with the meta-class CFlowJoinpoint, since pointcuts are used by cflow pointcuts. This association will allow modeling a cflow based pointcut that takes another pointcut as an argument.

5.3.10 Introduction

The operation of AsIntroductionAdvice of AspectS can be emulated by the static crosscutting feature, which is the introduction of AspectJ. Since introduction is an object in AspectS, like an advice, we decide to model the introduction as object too. In the profile, an introduction (static crosscutting) is modeled using the meta-class Introduction (Figure 5.12), which extends UML meta-class Class. Since static crosscutting features are used by aspects, the meta-class Introduction is associated with the meta-class Aspect.

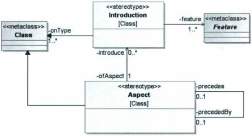


Figure 5.12: Introduction as a class extension

In order to specify which cross-cutting feature is to be introduced, the Introduction meta-class possesses a multi-valued attribute feature, whose data type is the UML meta-class Feature. onType is another multi-valued attribute of meta-class Introduction. It is introduced to specify the type on which the cross-cutting feature will be introduced. Since the same method can be introduced on multiple types, the attribute onType can have multiple values. As discussed earlier in Section 2.3.1.4 c) , in AspectS, each *AsIntroductionAdvice* object consists of a single *introBlock* that defines the body of a method to be introduced. As a result, if multiple methods on same or different datatypes (usually class in AspectS) are introduced using the same *AsIntroductionAdvice* object, all the methods will have the same body. In AspectJ, features can be introduced on both classes and interfaces. However, as AspectS does not have interfaces, we choose the UML meta-class Class as the data type of the attribute onType.

In Listing 5.2, an aspect A introduces a field *name* and a method *getName()* in AspectJ. The field *name* and the method *getName()* should be modeled respectively as an attribute and an operation of the associated aspect. When the profile is applied, the multi-valued attribute feature will be a tag definition of that aspect. For this example both *name* and *getName()* will be selected as the values of the tag feature. Since both of these features are to be introduced on a class named *Point*, the class *Point* should be selected as the value of tag onType from the base model.

Listing 5.2 Introducing a field and a method using AspectJ

```
aspect A {
    public String Point.name;
    public String Point.getName()  { return name; }
}
```

5.4 Profile for Static AOP

By combining the modeling elements described in previous section, we get the profile shown in Figure 5.13. This profile can be applied to the static AOP only. As in this profile, Joinpoint, JPDisjunction, Advice and Aspect are modeled as stereotype class; it will allow modeling the join points (Line# 2 and 3), composed join point (Line# 6), advice (Line# 13 and 14) and aspect (Line# 21) from the pseudocode shown in Listing 5.3. The listing shows the dynamic creation of joinpoint objects and adding these joinpoint objects to a dynamically created advice. Similarly, the advice object is added to the set of advices for an aspect object. It also shows the instantiation of Aspect along with their dynamic weaving.

Listing 5.3 Pseudocode for Dynamic AOP

```
1   //creating objects of JoinPoint.
2   x:= new jp1.
3   y:= new jp2.
4
5   //creating object of JPDisjunction
6   jpd:=JPDisjunction new.
7
8   //adding the joinpoint objects to the attributes of JPDisjuction.
9   jpd.joinpoints.add(jp1).
10  jpd.joinpoints.add(jp2).
11
12  //creating objects of Advice
13  ad1:= Advice new.
14  ad2:= Advice new.
15
16  //adding the JPDisjunction object to the attributes of Advice.
17  ad1.pointcut.add(jpd).
18  ad2.pointcut.add(jpd).
19
20  //creating object of AspectLogger
21  aspect:=AspectLogger new.
22
23  //adding the Advice objects to the attributes of AspectLogger.
24  aspect.advices.add(ad1).
25  aspect.advices.add(ad2).
26
27  //Aspect installation
```

```
28 aspect.install().
29
30 ......do something here.......
31
32 //Aspect uninstallation
33 aspect.uninstall().
```

However, there may be more than one instance of a joinpoint or advice specification, and that these instances are modifiable, assignable to variables and usable for method parameters. The profile does not support modeling join point objects or advice objects that can be assigned to variables (Line#9, 10, 17, 18, 24 and 25). Also, the above profile does not allow installing or uninstalling aspects (Line# 28 and 33). Hence, this profile does not support modeling dynamic weaving and dynamic AOP.

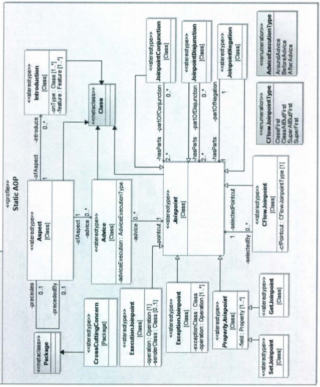


Figure 5.13: Profile for Static AOP

5.5 Profile for Dynamic AOP

We consider the profile shown in Figure 5.13 as a step towards the generic profile and modify it in this section to enable dynamic AOM. The following modeling elements are introduced to the profile for Static AOP:

a) Pointcut

In order to create an instance level connection between the meta-classes Advice and Joinpoint, the meta-class Pointcut that extends UML meta-class StructuralFeature is introduced to the previous profile (Figure 5.13).

We add the constraint that the «Pointcut» stereotype can only be applied to features of classes that are stereotyped «Advice». In other words, for all instances of a pointcut, the classifier of the pointcut feature must be an advice:

context Pointcut inv:

allInstances()→featuringClassifier.exists(C|C.oclIsKindOf(Advice))

We modify the association between Advice and Joinpoint meta-class shown in Figure 5.13. An advice object is associated with a pointcut; Pointcut meta-class (Figure 5.14) is associated with the Advice meta-class. Since each pointcut uses a join point or a set of join points that are composed with the meta-classes for join point composition, Joinpoint meta-class is associated with the Pointcut meta-class.

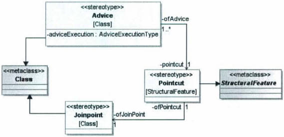


Figure 5.14: Pointcut as a structural feature extension

The above modification will allow modeling multiple instances of a joinpoint specification which are modifiable, assignable to variables and usable for method parameters. However this modification does not allow modeling a composed join point (Line# 17 and 18 of pseudocode shown in Listing 5.3).

b) JPCollection

With the meta-class Pointcut introduced in the previous Section, it is possible to model the passing of a single join point object, e.g. an execution join point(modeled using meta-class ExecutionJoinpoint) or an exception join point(modeled using the meta-class ExceptionJoinpoint) or a composed join point (modeled using the meta-class JoinpointDisjunction or the meta-class JoinpointConjunction). However, instances of a composed joinpoint specification that are modifiable, assignable to variables and usable for method parameters can not be modeled with the current profile. In order to model those there should be a connection between the meta-class Joinpoint with the meta-class JoinpointConjunction and JoinpointDisjunction. Hence, we cannot model Line# 9 and 10 of pseudocode shown in Listing 5.3 .

We introduce a new meta-class JPCollection (Figure 5.15) to the profile. Since join point collection uses joinpoint objects composed with the meta-classes for join point composition, JPCollection is associated with both JoinpointConjuction and JoinpointDisjunction.

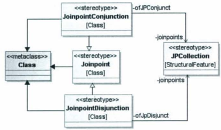


Figure 5.15: JPCollection as a structural feature extension

We add the constraints that the «JPCollection» stereotype can only be applied to features of classes that are either stereotyped «JoinpointConjunction» or «JoinpointDisjunction»:

context JPCollection inv:

allInstances()→featuringClassifier.exists(C|C.oclIsKindOf(JoinpointCojunction))

context JPCollection inv:

allInstances()→featuringClassifier.exists(C|C.oclIsKindOf(JoinpointDisjunction))

Since JoinpointConjunction and JoinpointDisjunction are associated with JPCollection meta-class, multiple instances of a composed join point can be modeled. JoinpointConjuction and JoinpointDisjunction do not need to be associated with Joinpoint meta-class (shown in Figure 5.13) anymore. We remove these associations.

The above modifications allow modeling a join point, which is composed of several join point objects (Line# 9 and 10 of pseudocode shown in Listing 5.3).

c) AdviceCollection

Same as join point specification, an advice specification can have multiple instances that are modifiable, assignable to variables, and usable for method parameters. A new meta-class AdviceCollection (Figure 5.16) that extends the UML meta-class StructuralFeature is introduced to the previous profile to model those instances. We modify the association between Aspect and Advice meta-class shown in Figure 5.13.

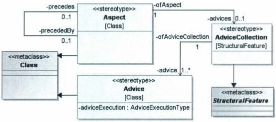


Figure 5.16: AdviceCollection as a structural feature extension

An aspect is associated with at most one advice collection; AdviceCollection meta-class is associated with the Aspect meta-class. Since each advice collection can consists of one or more advice objects, Advice meta-class should be associated with the AdviceCollection meta-class. We add the constraint that the «AdviceCollection» stereotype can only be applied to features of classes that are stereotyped «Aspect». In other words, for all instances of an advice collection, the classifier of the advice collection feature must be an aspect:

context AdviceCollection inv:

allInstances()→featuringClassifier.exists(C|C.oclIsKindOf(Aspect))

The above modification will allow modeling an advice specification with multiple instances (Line#24 and 25 of pseudocode shown in Listing 5.3) that are modifiable, assignable to variables, and usable for method parameters.

d) Install and Uninstall

In order to model the installation of an aspect, we introduce a meta-class install (Figure 5.17). The meta-class uninstall, as shown in Figure 5.17, is introduced to model the uninstallation of an aspect. Since installing and uninstalling the aspect are dynamic features that modify the behavior, we model the above meta-classes as stereotyped BehavioralFeature.

We add the constraint that the «install» stereotype can only be applied to operations of classes that are stereotyped «Aspect»:

context install inv:

allInstances()→featuringClassifier.exists(C|C.oclIsKindOf(Aspect))

Similarly, we add the constraint that the «uninstall» stereotype can only be applied to operations of classes that are stereotyped «Aspect»:

context uninstall inv:

allInstances()→featuringClassifier.exists(C|C.oclIsKindOf(Aspect))

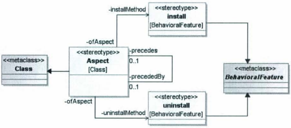


Figure 5.17: install and uninstall as behavioral feature extension

The above modifications will allow installing and uninstalling aspects (Line# 28 and 33 of pseudocode shown in Listing 5.3).

By modifying the profile for static AOP as mentioned in the above sections, we get the profile shown in Figure 5.18. However, as this dynamic profile excludes some of the relationships between elements of static profile, it can be applied to dynamic AOP only.

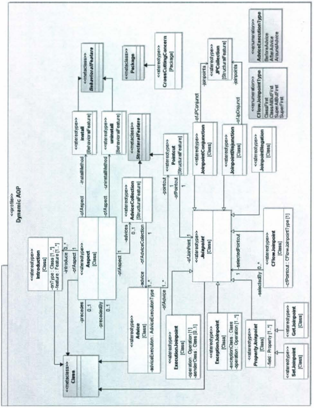


Figure 5.18: Profile for Dynamic AOP

5.6 Generic Profile

The profile developed in the previous section allows installing and uninstalling aspect. It also supports modeling multiple instances of join point and advice specification that these instances are modifiable, assignable to variables and usable for method parameters. However, it is only applicable to dynamic models. We consider this profile as another step towards the generic profile.

Recall that static AOP requires the developer to specify all pointcuts, advice and aspects at compile time. Usually a weaving compiler is used to add advice code to join points. As a result, aspects cannot be added, removed, or modified at runtime [16]. To change aspects, the system must be recompiled [17]. On the other hand, Dynamic AOP provides support for controlling aspects at runtime. It allows changes to aspects without restarting the program [16]. A run-time weaver is used to add advice code to the selected join points.

The approach (static or dynamic) that is followed by an AOP language extension can be understood by looking at the aspect weaving of that language. A language that uses a weaving compiler for aspect weaving follows the static AOP approach. On the other hand, runtime weaving of a language indicates that it follows the dynamic AOP approach. All the AOP extensions either follow static approach or dynamic approach but not at the same time. That is why, while modeling a system we should follow either of those approaches too. As a result, even if we come up with a single profile, the static and dynamic AOM should not be done at the same time.

The previous two profiles (static and dynamic) are individually appropriate for static AOM and dynamic AOM respectively. The generic profile includes all the modeling elements such as: Aspect, Advice, Joinpoint and Introduction that were in the profile for static AOP. As a result, it allows modeling the join points (Line# 2 and 3), composed join point (Line# 6), advice (Line# 13) and aspect (Line#19) from the pseudocode shown in Listing 5.3. The profile for static AOP was not having elements such as: AdviceCollection and Pointcut, which were introduced in the profile for dynamic AOP. These elements are included in the generic profile since they are necessary for dynamic AOP and also using those we still can model static AOP. On the other hand, the elements such as: JPCollection, install and uninstall should not be used for static AOM, since those are only used to model some specifications that represent dynamic AOP. As these elements are necessary for dynamic AOM, we keep these in the generic profile. As a result, it enables modeling multiple instances of a joinpoint and advice specification that are modifiable, assignable to variables and usable for method parameters.

Recall that, in the profile for dynamic AOP, the meta-class JPCollection was introduced to model a join point collection that possesses instances of multiple join points. Since join point collection uses joinpoint objects composed with the meta-classes for join point composition, JPCollection was associated with both JoinpointConjuction and JoinpointDisjunction. It allowed the modeling of multiple instances of a composed join point. As the associations of Joinpoint meta-class with JoinpointConjuction and JoinpointDisjunction became unnecessary, we removed these associations in the profile for dynamic AOP (Section 5.5 b)). However, in the generic profile, the meta-class JPCollection and its associations with JoinpointConjuction and JoinpointDisjunction are not used while modeling static AOP. As a result, the associations of Joinpoint meta-class with JoinpointConjuction and JoinpointDisjunction become vital. In the generic profile we restore these relationships (Figure 5.19).

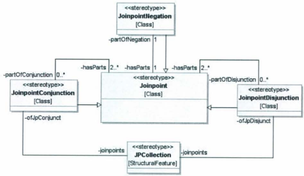


Figure 5.19: Associations with the meta-classes for join point composition

Figure 5.20 presents the final profile which is core generic meta-model for AOP languages. The generic profile is developed by combining modeling elements and their relationships from the previous two profiles (static and dynamic) in such a way that the role of the elements and their relationships present in the generic profile are significant to specify either static or dynamic AOP. As a result, the generic profile allows modeler to model static or dynamic AOP. The generic profile does not restrict a modeler to choose inappropriate elements, e.g. using JPCollection in the static model. However, the model will neither represent static AOP or dynamic AOP. In such case, if this model is further used for code generation, it will produce some incorrect and strange looking code. For this reason, it is the modeler's duty to make sure that static and dynamic AOM are not included within the same model.

Some of the applications of this profile are presented in next chapter.

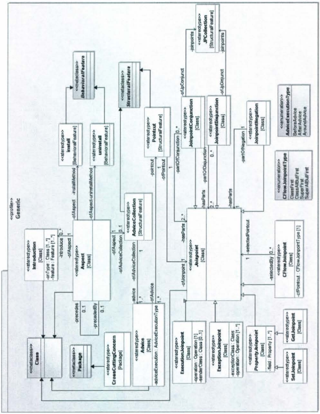


Figure 5.20: Core generic meta-model for AOP languages

5.7 Comparison with AspectJ profile

This section presents a comparative picture of our generic profile with the existing AspectJ profile [12]. Comparisons are done based on the modeling elements presented in AspectJ profile.

5.7.1 CrossCuttingConcern

In AspectJ, a cross-cutting concern contains aspects in the same way as packages contain classes. In the AspectJ profile [12], Evermann introduced the meta-class CrossCuttingConcern which extends the UML meta-class Package as a way of grouping related aspects. In the generic profile, we also use the meta-class CrossCuttingConcern to group related aspects of AOP languages.

5.7.2 Aspect

In the AspectJ profile, an aspect is modeled using the meta-class Class. In the generic profile, we also model aspect as class. However, some attributes of Aspect are omitted in the new profile. A Boolean attribute isPrivileged was introduced to indicate whether the aspect is privileged (discussed in Access Specification of Section 4.1.2). Since AspectS does not support access specification, we consider access specification as a feature specific to AspectJ. Hence, the Boolean attribute isPrivileged of AspectJ profile is absent in the generic profile. Similarly, the attribute declaredImplements that allows the declaration of interface realizations is omitted.

In AspectJ, aspects may be instantiated per pointcut. As shown in Figure 5.21, the attributes perType and perPointCut that specify the type of aspect instantiation and associated pointcut per pointcut were introduced in the AspectJ profile. These attributes are also omitted because the related features are AspectJ specific. Since perType is not kept in generic profile, the values provided to it by the enumeration AspectInstanceType cannot be kept in the new profile as well.

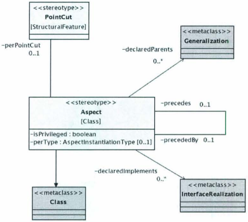


Figure 5.21: Aspect as a class extension in AspectJ profile[12]

However, the new profile includes Aspect precedence from AspectJ profile without any change.

5.7.3 Advice

In the AspectJ profile, with Aspect being a meta-class that extends Class, the dynamic features of aspects, i.e. advices, play the role of class behavior. That is why the meta-class Advice was modeled as an extension of the meta-class BehavioralFeature. In order to allow modeling advice objects that can be passed as method parameter, in the generic profile, we model the meta-class Advice as an extension of the meta-class Class.

The attribute adviceExecution of Advice meta-class is reused in the generic profile without any modification. Similar to the AspectJ profile, in the generic profile, the types of an advice are also modeled using the meta-class AdviceExecutionType that extends the UML meta-class Enumeration.

5.7.4 PointCut

In AspectJ profile, a pointcut is modeled using meta-class StructuralFeature. In AspectS, pointcut is a collection of join points. Since join points are objects in AspectS, they should be modeled as objects. PointCut meta-class (from AspectJ profile) is renamed as Joinpoint that extends the UML meta-class Class in generic profile.

5.7.5 OperationPointCut

It is a superclass to describe pointcuts that select operation related join points. In the new profile, execution pointcut (Section 5.3.6) is the only operation related join point.

5.7.5.1 ExecutionPointCut

In the AspectJ profile, a pointcut that selects method execution join point was modeled using meta-class StructuralFeature. We rename it as ExecutionJoinpoint. As discussed earlier in this chapter (Section 5.3.6), in the new profile, the meta-class ExecutionJoinpoint extends UML meta-class Class.

In the AspectJ profile, PreInitializationPointCut and InitializationPointCut were modeled as subclasses of ExecutionPointCut meta-class. As discussed earlier in chapter 4 (Section 4.1.1e) , Section 4.1.1f) and Section 4.1.1g)), these elements are based on the features specific to AspectJ. As a result, we do not keep those in the new profile.

5.7.5.2 CallPointcut

In AspectJ profile, a pointcut that selects method call join point was modeled using meta-class StructuralFeature. As discussed in Section 5.3.6, CallPointcut is not kept in the new profile.

5.7.5.3 WithinCodePointCut

The meta-class WithinCodePointCut of AspectJ profile is not kept in new profile since AspectS does not allow such join point selection (Page 90).

5.7.6 PointCutPointCut

In the AspectJ profile, the meta-class PointCutPointCut is a superclass for the meta-classes CFlowPoint and CFlowBelowPoint that models *CFlow* and *CFlowBelow* pointcuts of AspectJ respectively. In order to unify control flow related pointcuts of AspectJ and AspectS, we introduce the meta-class CFlowJoinpoint that extends UML meta-class Class. The attribute cfPointcut specifies different types of control flow related pointcuts from both the languages. Since we only use a single meta-class CFlowJoinpoint to model the control-flow based join point, we remove the super-class PointCutPointCut from the new profile.

5.7.7 AdviceExecutionPointCut

In the AspectJ profile, a pointcut that selects advice execution as join point was modeled using meta-class AdviceExecutionPointCut, which extends UML meta-class StructuralFeature. Since this is specific to AspectJ (Section 4.1.1h)), this element is not included in the new profile.

5.7.8 PropertyPointCut

The meta-class PropertyPointCut that extends UML meta-class StructuralFeature is an abstract meta-class. In the generic profile, this element renamed as PropertyJoinpoint with a modified extension; it extends UML meta-class Class.

In AspectJ profile the subclasses of PropertyPointCut such as GetPointCut, and SetPointCut that were modeled extending the UML meta-class Structural Feature, are reused with modifications. These are renamed as GetJoinpoint and SetJoinpoint respectively (Section 4.1.1c)). Both of these meta-classes extend the UML meta-class Class.

5.7.9 ContextExposingPointCut

The meta-class ContextExposingPointCut extends the UML meta-class StructuralFeature. It is an abstract superclass of those pointcuts that expose context in advices [12]. Exposing context in an advice is implicit to AspectS (Page 91). Also, for AspectJ, it can be handled during code generation. Hence, we do not include it as a modeling element in the new profile.

5.7.10 TypePointCut

In the AspectJ profile, the meta-class TypePointCut extends the UML meta-class StructuralFeature. It is a superclass to model pointcuts that select type-related join points [12]. The subclasses of the meta-class TypePointCut: ThisPointCut, ArgsPointCut, TargetPointCut, WithinPointCut, and StaticInitializationPointCut were modeled based on the features specific to AspectJ. Thus, these meta-classes are ignored in the generic profile. However, join point arguments are automatically exposed to advices in AspectS. As a result, for each execution join point selection, an args pointcut is created during code generation.

In the AspectJ profile, the meta-class ExceptionPointcut extends the UML meta-class StructuralFeature. We model exception join point using the meta-class ExceptionJoinpoint that extends the UML meta-class Class.

5.7.11 Pointcut composition

PointCutConjunction, PointCutDisjunction, and PointCutNegation were introduced in AspectJ profile to model the composition of pointcuts. The same idea is followed in the generic profile to model the composition of join points. Since Joinpoint is stereotyped classes, the meta-classes for join point compositions are also modeled as stereotyped Class, where as those were modeled as stereotyped StructuralFeature in AspectJ profile.

5.7.12 StaticCrossCuttingFeature

In the AspectJ profile, the static crosscutting feature was modeled using the meta-class StaticCrossCuttingFeature that extends UML meta-class Feature. We model the static crosscutting feature with the meta-class Introduction. In the generic profile, the meta-class Introduction extends UML meta-class Class.

5.8 Summary

Similar to the AspectJ profile developed in [12], the present work allows the integration of aspect features with base-model features on the meta-model level, rather than as part of the model. We define all the elements as UML stereotypes, tags on those stereotypes or as the values of tags. A UML stereotype is a meta-class which enters into extends relationships with existing meta-classes [28]. Visually, this is shown with the extended class in square brackets. Attributes that are modeled on stereotype meta-classes will translate to tags when the profile is applied [28]. Similarly, values of stereotype attributes will become values of tags when the profile is applied [28]. This extension mechanism in UML2.0 is therefore a powerful way in which any meta-level model immediately becomes usable as a profile.

We develop the profiles for static and dynamic AOP separately as steps towards the generic profile. The profile for Static AOP allows modeling aspects, advices and join points as stereotyped Classes. However, it can be applied to the static models only. Some modifications to this profile such as: including modeling elements for enabling and disabling aspects, and introducing meta-classes such as: AdviceCollection, Pointcut and JPCollection as Structural feature, gives us the Profile for Dynamic AOP.

Profile for Dynamic AOP allows modeling the join point and advice objects that can be passed as methods arguments. Since this modified profile excludes some relationships from the profile for Static AOP, we consider this profile for dynamic AOM only.

We combine the modeling elements from both profiles (static and dynamic). It gives us the generic profile, which allows modeler to model both static and dynamic AOP with the constraint that modeler is modeling either static or dynamic AOP but not both at the same time.

Since this generic profile is an extension of [12], some modeling elements, as shown in Table 5.2, are reused from the AspectJ profile without modification.

Table 5.2: Reusing elements from AspectJ Profile without modification

Elements	Modeled As
Crosscutting Concern	Package
AdviceExecutionType	Enumeration

However, to allow dynamic AOP modeling, as shown in Table 5.3, most of the existing elements of [12] were modeled by providing appropriate extensions.

Table 5.3: Reusing elements from AspectJ Profile with modification

AspectJ Profile		Generic Profile	
Elements	**Modeled As**	**Elements**	**Modeled As**
Aspect	Class	Aspect	Class
PointCut	Structural Feature	Joinpoint	Class
Advice	Behavioral Feature	Advice	Class
PointCutConjunction	Structural Feature	JoinpointConjunction	Class

AspectJ Profile		Generic Profile	
Elements	**Modeled As**	**Elements**	**Modeled As**
PointCutDisjunction	Structural Feature	JoinpointDisjunction	Class
PointCutNegation	Structural Feature	JoinpointNegation	Class
ExecutionPointCut	Structural Feature	ExecutionJoinpoint	Class
PropertyPointCut	Structural Feature	PropertyJoinpoint	Class
GetPointCut	Structural Feature	GetJoinpoint	Class
SetPointCut	Structural Feature	SetJoinpoint	Class
ExceptionPointcut	Structural Feature	ExceptionJoinpoint	Class
CFlowPointCut	Structural Feature	CFlowJoinpoint	Class
CFlowBelowPointCut	Structural Feature		
StaticCrossCuttingFeature	Feature	Introduction	Class

We know that AspectJ is very rich for its pointcut constructs that allows wide varieties of join point selections. The AspectJ profile takes account of those pointcuts and provides full facility for modeling them. Our profile is based on the core generic features of different AOP languages. Since the languages other than AspectJ has limited pointcut constructs comparing to that of AspectJ, several pointcuts from the AspectJ profile, as shown in Table 5.4, are omitted in the generic profile.

Table 5.4: Elements omitted from AspectJ Profile

AspectJ Profile		Reason for excluding
Elements	**Modeled As**	
AspectInstantiationType	Enumeration	Specific to AspectJ
AdviceExecution Pointcut	Structural Feature	

AspectJ Profile		Reason for excluding
Elements	**Modeled As**	
OperationPointCut	Structural Feature	It is a superclass to describe pointcuts that select operation related join points. In the new profile, execution pointcut (Section 5.3.6) is the only operation related join point. Hence, OperationPointCut is ignored.
CallPointCut	Structural Feature	Specific to AspectJ
PreInitializationPointCut	Structural Feature	
InitializationPointCut	Structural Feature	
WithinCodePointCut	Structural Feature	
ContextExposingPointCut	Structural Feature	Implicit in AspectS. It can be handled during code generation for both AspectS and AspectJ.
TypePointCut	Structural Feature	AspectS is untyped. TypePointCut is specific to AspectJ.
ThisPointCut	Structural Feature	Specific to AspectJ
TargetPointCut	Structural Feature	
StaticInitializationPointCut	Structural Feature	
WithinPointCut	Structural Feature	
ArgsPointCut	Structural Feature	
PointCutPointCut	Structural Feature	PointCutPointCut is a superclass for the control-flow based pointcuts of AspectJ. We use a single meta-class CFlowJoinpoint to model the control-flow based join point. Hence, PointCutPointCut is omitted in the generic profile.

In order to allow dynamic AOM, following modeling elements are introduced in the generic profile (Table 5.5).

Table 5.5: New elements in the Generic profile

Elements	Modeled As
install	Behavioral Feature
uninstall	Behavioral Feature
AdviceCollection	Structural Feature
Pointcut	Structural Feature
JPCollection	Structural Feature
CFlowJoinpointType	Enumeration

Chapter 6

APPLICATION EXAMPLE

As discussed earlier, the proposed profile can be applied to both static and dynamic AOP at different times. In this chapter, we show applications of the proposed profile as proof of concepts. Rather than using complex case studies, we illustrate some simple examples that are already discussed in previous chapters. Using those examples, we demonstrate the use of the profile during modeling and show the visual appearance of the model. As the profile does not allow modeling of static and dynamic AOP at the same time, modeling crosscutting concerns for static and dynamic AOP implementations will be different for all examples. In Chapter 3, we discussed the difference between static and dynamic AOP. We have seen that AspectJ and AspectS follow static and dynamic AOP approaches, respectively. In the next few sections, for each modeling example, we will consider AspectJ code to develop the static model. On the other hand, while modeling the dynamic aspects, AspectS code will be taken into account.

6.1 Example-1 : Modeling SenderClassSpecific Join Point

Recall Section 5.3.6, where we introduced the element ExecutionJoinpoint to model both execution and call join points. Modeling an execution joinpoint is trivial. The following paragraphs present an example of senderClassSpecific pointcut in AspectJ.

The base model is same for both static and dynamic AOP implementations. Section 6.1.2.1 and Section 6.1.2.2 respectively present static and dynamic models of cross-cutting concerns for this example.

6.1.1 Base Model

This section presents the base model of the example. The core concerns involve two classes *Test* and *NewTest*. The classes and their relationships are shown using the UML class diagram shown in Figure 6.13. We wish to advise the call of the *deliver* method by objects of class *NewTest*.

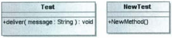


Figure 6.1: Base model of the example with call join point

6.1.2 Crosscutting-cutting Concern

Section 6.1.2.1 and Section 6.1.2.2 present the cross-cutting concern of this senderClassSpecific join point example in terms of static and dynamic models.

6.1.2.1 Modeling Static AOP

In Listing 6.1, the aspect *AfterHelloWorld* is defined within the package *CrossCuttingConcern*. The aspect *AfterHelloWorld* consists of a pointcut and an after advice that advises the pointcut *deliverMessage*. The pointcut *deliverMessage* selects the call to the *deliver* method of the *Test* class from the caller *NewTest* class. Both the classes Test and NewTest, along with their methods, are shown in the base model (Figure 6.1).

Listing 6.1 An example of a call pointcut in AspectJ

```
package CrossCuttingConcern;
public aspect AfterHelloWorld {
pointcut deliverMessage(String message): call (* Test.deliver(..))
                                 && this(NewTest)&& args(message);
after(String message): deliverMessage(message){
      System.out.println();
     }
}
```

Recall that in UML, meta-classes that extend existing meta-classes become stereotypes, and attributes of extending meta-classes become tags. Crosscutting concerns become packages that are stereotyped «CrossCuttingConcern» and the aspects of this cross-cutting concern are classes that are stereotyped «Aspect», contained in the package.

In the above example, the pointcut signature of *deliverMessage* indicates that a method call from a specific class is selected. As a result, the pointcut should be modeled as an «ExecutionJoinpoint». Moreover, as the call is from the class *NewTest*, using the attribute senderClass, the class *NewTest* should be specified as the sender class. We will not model the args() pointcut as that only exposes the context and can be automatically generated for each execution pointcut during code generation.

The advice should be modeled as an «Advice». Since this is an after advice, the value of the tag adviceExecution should be selected as AfterAdvice. Figure 6.2 represents the static model of the senderClassSpecific join point example.

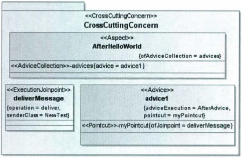


Figure 6.2: Static model of senderClassSpecific join point example

6.1.2.2 Modeling Dynamic AOP

In this section we will model the dynamic implementation (in AspectS) of the senderClassSpecific join point example. Listing 6.2 presents the aspect *AfterHelloWorld* in AspectS. The category in the class description shows that this aspect belongs to the category *CrossCuttingConcern*, which should be modeled as a package.

Listing 6.2 *AfterHelloWorld* aspect in AspectS

```
AsAspect subclass: #AfterHelloWorld
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CrossCuttingConcern'
```

Listing 6.3 shows a method returning an advice of the *AfterHelloWorld* aspect. The *afterBlock* of the advice indicates that the kind of this advice is AfterAdvice.

Listing 6.3 advice of *AfterHelloWorld* aspect

```
advice1
  ^ AsBeforeAfterAdvice
      qualifier: (AsAdviceQualifier
          attributes: {#senderClassSpecific})
      pointcut: [{AsJoinPointDescriptor
          targetClass: Test targetSelector: #deliver}]
      afterBlock: [:receiver :arguments :aspect :client :return|
                       Transcript show: '  Goodbye World.  '.]
```

The advice qualifier attribute in this listing provides us the information about the type of join point to be modeled. With the attribute *senderClassSpecific*, it indicates that the join point is an execution join point that contains sender class information. From the description of the pointcut we get the information of the operation to be selected as a join point. Listing 6.4 indicates how the definitions in Listing 6.3 can be used. The class *NewTest* is added as a sender class to the aspect (Line#2).

Listing 6.4 Dynamic weaving in AspectS

```
aspect :=AfterHelloWorld new.
test1 := NewTest new.
aspect addSenderClass: NewTest.
aspect install.
test1 newMethod.
aspect removeSenderClass: NewTest.
aspect uninstall.
```

As the elements aspect, advice collection, advice, pointcut, and join point of this dynamic implementation are similar to that of static implementation (Listing 6.1), these elements can be modeled exactly as shown in Figure 6.2.

However, dynamic aspect weaving is not supported by the languages that follow static adaptation. As a result, the static model shown in the previous section does not include any element related to dynamic aspect weaving. Line# 4 and Line#7 of Listing 6.4

presents dynamic weaving that controls aspect installation during runtime. Installation of aspects is related to behavior. As a result, it should be modeled as operation. To model enabling and disabling of aspect, as shown in Figure 6.3, we create two operations *installMe* and *uninstallMe* for *AfterHelloWorld*. When the profile is applied, these operations become stereotyped «install» and «uninstall» respectively. We select *installMe* and *uninstallMe* as values of the installMethod and uninstallMethod tags of AfterHelloWorld.

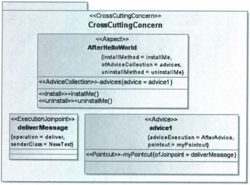


Figure 6.3: Dynamic model of senderClassSpecific join point example

6.2 Example-2 : Modeling Cflow Join Point

In Chapter 4 (Section 4.1.4b) ii), we showed examples based on the control flow based join points. In this section, we use one of those examples to show the application of the element CFlowJoinpoint of the generic profile.

6.2.1 Base Model

This section presents the base model (Figure 6.4) of the example with cflow join point. The core concerns involve a single class *AsFactorialM* as shown in the following UML class diagram.

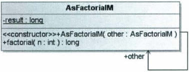


Figure 6.4: Base model of CFlow Join Point Example

6.2.2 Cross-cutting Concern

Assume that we want to advise all recursive calls to factorial() except the top one. In this example, the advice is triggered every time except for the first time when the method *factorial* is invoked by the instance of *AsFactorialM* class. Let us assume that the aspect *AspectClassAllButFirst* is defined within the package *CrossCuttingConcern*. The aspect *AspectClassAllButFirst* is associated with the advice collection advices, which is a collection of a single after advice Advice1. The advice Advice1 advises the pointcut myPointcut1, which is a collection of the join point Pointcut2. The join point Pointcut2 is the control flow based pointcut. As a result, it takes the join point Pointcut1, which is an execution join point as an argument. Section 6.2.2.1 and Section 6.2.2.2 respectively present the static and the dynamic models of this cflow join point example.

6.2.2.1 Modeling Static AOP

Modeling the elements package, aspect, advice collection, advice, and pointcut is similar to the previous example. However, as the join point to be advised restricts the set of execution pointcuts to all but the first one, it should be modeled using the meta-class «CFlowJoinpoint». Also, ClassAllButFirst should be selected as the value of the tag cfPointcut of «CFlowJoinpoint». Since cflow related pointcuts restrict a set of join points, they take another pointcut as argument. In this case, the control flow pointcut is intended to restrict the set of execution pointcuts to all but the first one. Therefore, cflow pointcuts must refer to the set of pointcuts that they restrict. A join point related to the latter should be modeled extending the meta-class «ExecutionJoinpoint». This «ExecutionJoinpoint» should be selected as the value of the selectedPointcut tag of «CFlowJoinpoint». The following paragraph provides a detailed description of modeling these two join points.

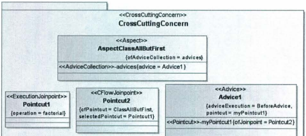


Figure 6.5: Static cross-cutting concern of cflow join point example

As shown in Figure 6.5, we create two classes *Pointcut1* and *Pointcut2* for the two join points mentioned above. We apply the stereotype «ExecutionJoinpoint» on *Pointcut1*. The attribute *operation* of «ExecutionJoinpoint» becomes a tag that provides a list of operations whose execution can be selected as join points. We select the method *factorial* of *AsFactorialM* class from the base model (Figure 6.4) as the value of the operation tag of Pointcut1. Pointcut1 represents the set of execution join points to be selected and advised, but this set is further qualified when Pointcut1 is linked to the control flow pointcut, Pointcut2.

The stereotype «CFlowJoinpoint» is then applied to the class *Pointcut2*. Its attributes cfPointcut and selectedPointcut become tags. The tag cfPointcut specifies the type of the control flow based join point to be modeled. The tag selectedPointcut allows connecting Pointcut2 with another pointcut. We select ClassAllButFirst and Pointcut1 as the values of cfPointcut and selectedPointcut respectively.

6.2.2.2 Modeling Dynamic AOP

While modeling the dynamic cross-cutting concerns of the cflow join point, besides modeling all the elements modeled in the static model, we need to model the dynamic weaving of aspect.

As shown in Figure 6.6, at first we model all the elements such as aspect, advice collection, advice, pointcut, and join points by following the procedure discussed for the static model in Section 6.2.2.1 . Then, to model enabling and disabling of aspect, we create two operations *installMe* and *uninstallMe* for AspectClassAllButFirst. When the profile is applied, like in the dynamic model of the previous example (Section 6.1.2.2), these operations become stereotyped «install» and «uninstall» respectively, which are then selected as values of installMethod and uninstallMethod of AspectClassAllButFirst.

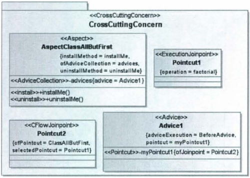


Figure 6.6: Dynamic cross-cutting concern of cflow join point example

6.3 Example-3 : Modeling an Exception Join Point

Recall Chapter 4 (Section 4.1.1d)), where we showed an example of the exception handler execution join point. In this section, we use that example to show the application of the element ExceptionJoinpoint of the generic profile.

6.3.1 Base Model

This section presents the base model (Figure 6.7) of the example with an exception join point. The core concerns involve two classes *TestHandler* and *Error* as shown the following UML class diagram. Assume that we wish to advise the execution of the handlers that handle exceptions of class *Error* that are raised by the method *deliver()*.

Figure 6.7: Base model of exception join point example

6.3.2 Crosscutting-cutting Concern

In this example, within a *CrossCuttingConcern* the aspect *AspectHandler* consists of a pointcut deliverMessage and a before advice. An exception of type *Error* that is raised by the *deliver* method of the *TestHandler* class is selected as a join point by the pointcut deliverMessage, which is advised by the before advice Advice1. Section 6.3.2.1 and Section 6.3.2.2 respectively present the static and the dynamic models of this exception join point example.

6.3.2.1 Modeling Static AOP

Modeling the elements package, aspect, and advice is similar to the first example (Section 6.1.2.1). However, as the join point to be advised involves the occurrence of an exception within a method, it should be modeled using the meta-class «ExceptionJoinpoint».

Since the meta-class ExceptionJoinpoint is a subclass of the Joinpoint meta-class, as shown in Figure 6.8, the class *deliverMessage* becomes stereotyped «ExceptionJoinpoint». Its attribute *operation* becomes a tag that specifies an operation in which the exception will be occurred. We select the method *deliver* of *TestHandler* class from the base model (Figure 6.7) as the value of operation of deliverMessage, indicating that we wish to select exceptions raised by this operation. Its other attribute exceptionClass also becomes a tag that specifies the type of the exception whose handling we wish to advise. The class *Error* from the base model is selected as the value of exceptionClass.

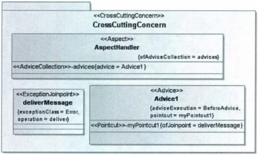


Figure 6.8: Static cross-cutting concern of cflow join point example

6.3.2.2 Modeling Dynamic AOP

While modeling the dynamic cross-cutting concerns of the exception join point, unlike the previous examples, we follow a different way that clearly shows the modeling of the join point objects that can be instantiated and assigned to variables. However, these two approaches are equivalent in terms of the code being generated.

In this approach, we introduce a class *CollectionType*, e.g. a Java collection type that will collect the advice instances or objects of an aspect. This class is associated with the aspect and the advice.

As shown in Figure 6.9, we model all the elements such as aspect, join points, and advice as described in previous section. Since Advice1 is associated with the class

CollectionType (objects in the collection are of instances of Advice1), the attribute *myPointcut* of Advice1 becomes the stereotyped «Pointcut». The value of ofJoinpoint of myPointcut is set to deliverMessage, indicating that we wish to collect advice to exception handlers for exceptions raised by the *deliver()* operation. Also the value of pointcut of Advice1 is set to myPointcut so that advice objects "know" about the collection they are collected in.

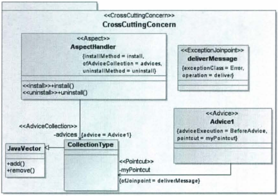


Figure 6.9: Dynamic cross-cutting concern of exception join point example

AspectHandler is associated with the class *CollectionType* by its attribute *advices* which is stereotyped «AdviceCollection» to indicate that this field will contain a collection of advices. The meta-attribute *advice* becomes a tag. Advice1 is selected as the value of the tag because these are the types of advices we wish to collect in the advice collection.

Like the previous dynamic models, we create two operations *install* and *uninstall* for AspectHandler. When the profile is applied, these operations respectively become stereotyped «install» and «uninstall», which are then selected as values of installMethod and uninstallMethod of AspectHandler.

6.4 Example-4 : Modeling a Property Join Point

Recall Chapter 4 (4.1.1c)), where we showed an example of field access join point. In this section, we use that example to show the application of the element PropertyJoinpoint of the generic profile.

6.4.1 Base Model

This section presents the base model (Figure 6.10) of the example with the property join point. The core concerns involve a single class *AsCounterModified1* as shown in the following UML class diagram.

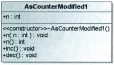


Figure 6.10: Base model of property join point example

6.4.2 Crosscutting-cutting Concern

Section 6.4.2.1 and Section 6.4.2.2 respectively present the static and the dynamic models of this exception join point example.

6.4.2.1 Modeling Static AOP

In this section, we will model the pointcuts that select read and write access of field as join points. In Listing 6.5, the aspect *AspectLogger* is defined within the package *CrossCuttingConcern*. It consists of two pointcuts and two before advices. The pointcut getN selects the read access of the field *n* of the *AsCounterModified1* class and advised by the first advice. Similarly, the pointcut setN selects the write access to the field *n* of and advised by the second advice.

Listing 6.5 An example of field access pointcuts in AspectJ

```
package CrossCuttingConcern;
public aspect AspectLogger {
      pointcut getN(): get(public int AsCounterModified1.n);
      pointcut setN(): set(public int AsCounterModified1.n);
      before() : getN() {}
      before() : setN() {}
}
```

As shown in Figure 6.11, modeling the elements package and aspect is similar to the first example (Section 6.1.2.1). However, the pointcuts to be advised are field access pointcuts, which should be modeled using the subclasses of the meta-class «PropertyJoinpoint».

We model two classes *getN* and *setN*. When the profile is applied, *getN* and *setN* become stereotyped «GetJoinpoint» and «SetJoinpoint» respectively. Their inherited attribute *field* becomes a tag that specifies the field whose access will be selected as join point. We select the field *n* of the *AsCounterModified1* class from the base model (Figure 6.10) as the values of field of getN and setN, indicating that we wish to select the read and write access of the field n.

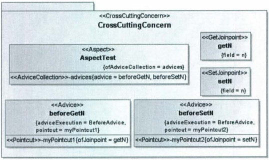


Figure 6.11: Static cross-cutting concern of property join point example

The classes *beforeGetN* and *beforeSetN* become stereotyped «Advice». The values of adviceExecution of beforeGetN and beforeSetN are set to BeforeAdvice. The value of pointcut of beforeGetN is set to myPointcut1 so that the advice "knows" about the collection of join points to be advised. Similarly, the value of pointcut of beforeSetN is set to myPointcut2. The values of ofJoinpoint of myPointcut1 and myPointcut2 are set to getN and setN respectively indicating the members of join point collection to be advised.

The meta-attribute advices of AspectTest becomes stereotyped «AdviceCollection». Both beforeGetN and beforeSetN are selected as the values of advices to indicate that this field will contain a collection of advices.

6.4.2.2 Modeling Dynamic AOP

While modeling the dynamic cross-cutting concerns we consider the emulated field access pointcut (Listing 6.6) of AspectJ in AspecS. We follow the approach presented in the previous example. As before, we introduce a class *CollectionType*, which is associated with the aspect and the advice.

Listing 6.6 Emulated AspectJ's field access pointcuts in AspectS

```
AsAspect subclass: #AspectTest
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'CrossCuttingConcern'!

!AspectTest methodsFor: 'as yet unclassified' stamp: ''!
adviceGetField
     ^ AsBeforeAfterAdvice
     qualifier: (AsAdviceQualifier
      attributes: {#receiverClassSpecific})
     pointcut: [{AsJoinPointDescriptor
      targetClass: AsCounterModified1 targetSelector: #n}]
     beforeBlock: [:receiver :arguments :aspect :client |]! !

!AspectTest methodsFor: 'as yet unclassified' stamp: ''!
adviceSetField
     ^ AsBeforeAfterAdvice
     qualifier: (AsAdviceQualifier
      attributes: {#receiverClassSpecific})
     pointcut: [{AsJoinPointDescriptor
      targetClass: AsCounterModified1 targetSelector: #n:}]
     beforeBlock: [:receiver :arguments :aspect :client |]! !
```

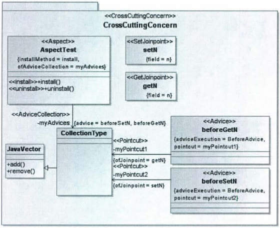


Figure 6.12: Dynamic cross-cutting concern of property join point example

As shown in Figure 6.12, we model all the elements such as package, aspect, and join points similar to the static model (Figure 6.11). The other elements that are related to advice collection, pointcuts, and dynamic weaving are modeled as described in the dynamic model of the previous example (Section 6.3.2.2).

The elements presented in the above model can be used as shown in the pseudo code in Listing 6.7.

Listing 6.7 Pseudo code that uses the elements presented in Figure 6.12

```
//creating objects of JoinPoint.
x:= new getN.
y:= new setN.

//creating objects of Advice
ad1:= beforeGetN new.
ad2:= beforeSetN new.
```

```
//adding the JoinPoint objects to the attributes of Advice.
ad1.myPointcut1.add(x).
ad2.myPointcut2.add(y).

//creating object of AspectTest.
aspect:=AspectTest new.

//adding the Advice objects to the attributes of AspectTest.
aspect.myAdvices.add(ad1).
aspect.myAdvices.add(ad2).

//Aspect installation
aspect.install().

......do something here.......

//Aspect uninstallation
aspect.uninstall().
```

6.5 Example-5 : Modeling the Shopping-Cart Example

Recall Chapter 2 (Section 2.2.2 and Section 2.3.2), where we showed the implementation of the shopping-cart example as running examples in AspectJ and AspectS respectively. In this section, we use the shopping-cart example to show the application of the composed join points of the generic profile.

6.5.1 Base Model

This section presents the base model (Figure 6.13) of the shopping-cart example. The main classes (core concerns) and their relationships are shown using UML class diagram.

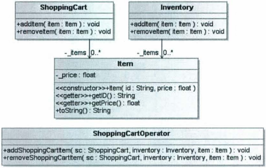


Figure 6.13: Base model of shopping-cart example

6.5.2 Crosscutting-cutting Concern

Section 6.5.2.1 and Section 6.5.2.2 respectively present the static and the dynamic models of this shopping-cart example.

6.5.2.1 Modeling Static AOP

In this section, we will model composed pointcuts of AspectJ. In Listing 6.8, the aspect *TraceAspect* is defined within the package *CrossCuttingConcern*. The aspect *TraceAspect* consists of two composed pointcuts JPDisjunction1 and JPDisjunction2. The pointcut JPDisjunction1 is composed of execution pointcuts jp1, jp2, jp3, and jp4. On the other hand, JPDisjunction2 is composed of execution pointcuts jp5 and jp6. Pointcuts JPDisjunction1 and JPDisjunction2 are advised by two before advices.

While modeling this example, each of the execution join points should be modeled as stereotyped «ExecutionJoinpoint». We will again ignore modeling the args pointcuts, since those merely expose context and are automatically generated during code generation. Two composed join points should be modeled as stereotyped «JoinpointDisjunction». Modeling the other elements such as package, aspect, advice collection, advice, and pointcut is same as the previous static models. The following paragraph presents a detailed description of modeling the above mentioned join points.

Listing 6.8 The shopping-cart example in AspectJ

```
package CrossCuttingConcern;

public aspect TraceAspect {

    pointcut jp1(Item item):
          execution(* Inventory.addItem(..))&& args(item);

    pointcut jp2(Item item):
          execution(* Inventory.removeItem(..))&& args(item);

    pointcut jp3(Item item):
          execution(* ShoppingCart.addItem(..))&& args(item);

    pointcut jp4(Item item):
          execution(* ShoppingCart.removeItem(..))&& args(item);

    pointcut jp5(ShoppingCart sc,Inventory inventory,Item item):
          execution(*ShoppingCartOperator.addShoppingCartItem(..))
          && args(sc,inventory,item);

    pointcut jp6(ShoppingCart sc,Inventory inventory,Item item):
          execution(*ShoppingCartOperator.addShoppingCartItem(..))
          && args(sc,inventory,item);

     pointcut JPDisjunction1(Item item)
     : jp1(item)||jp2(item)||jp3(item)||jp4(item);


    pointcut JPDisjunction2(ShoppingCart sc, Inventory inventory,
                                                          Item item)
     : jp5(sc,inventory,item)||jp6(sc,inventory,item);
```

```
before(Item item) : JPDisjunction1(item) {}
before(ShoppingCart sc, Inventory inventory, Item item) :
                      JPDisjunction2(sc,inventory,item) {}

}
```

As before, join points are defined as meta-class extensions of the Class meta-class. Since ExecutionJoinpoint is a subclass of the Joinpoint meta-class, the *jp1*, *jp2*, *jp3*, *jp4*, *jp5*, and *jp6* classes become stereotyped «ExecutionJoinpoint». The attribute *operation* becomes a tag that provides a list of operations whose execution can be selected as join points.

We select the method *addItem* of the *Inventory* class from the base model as the value of operation of jp1 indicating that the execution of the method *addItem* will be selected as a join point. Similarly, the method *removeItem* of the class *Inventory*, the method *addItem* of the class *ShoppingCart*, the method *removeItem* of the class *ShoppingCart*, the method *addShoppingCartItem* of the class *ShoppingCartOperator*, and the method *removeShoppingCartItem* of the class *ShoppingCartOperator* are selected as the values of operation of jp2, jp3, jp4, jp5 and jp6 respectively.

Since JoinpointDisjunction is a subclass of the Joinpoint meta-class, the classes *JPDisjunction1* and *JPDisjunction2* are also stereotyped as «JoinpointDisjunction». The attribute hasParts of stereotyped «JoinpointDisjunction» specifies join points that are parts of composition. A modeler must make sure that the number and type of arguments are consistent for all the join points that are part of a join point disjunction or join point conjunction. For example, jp1 and jp2 can be parts of the same composed join point since their argument number (a single argument item) and type (

the type of item is Item) are the same. However, jp1 and jp5 cannot be parts of a same composed join point since their argument number and type are different.

We choose the join points jp1, jp2, jp3, and jp4 as the values of hasParts of JPDisjunction1 so that their composition can be selected as a join point. Similarly, jp5 and jp6 are chosen as the values of hasParts of JPDisjunction2.

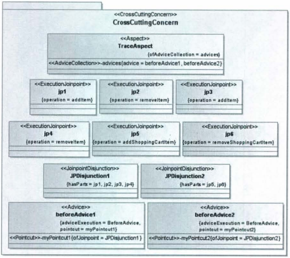


Figure 6.14: Static cross-cutting concern of the shopping-cart example

6.5.2.2 Modeling Dynamic AOP

While modeling the dynamic cross-cutting concerns we consider the emulated composed pointcut of AspectJ in AspectS. We follow the approach presented in the previous two examples.

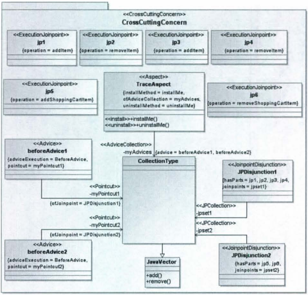


Figure 6.15: Dynamic cross-cutting concern of the shopping-cart example

As shown in Figure 6.15, we model all the elements such as package, aspect, and join points similar to the static model (Figure 6.14). The other elements that are related to advice collection, pointcuts, and dynamic weaving are modeled as described in the dynamic model presented in Section 6.3.2.2 .

The elements presented in the above model can be used as show in the pseudo code in Listing 6.9.

Listing 6.9 Pseudo code that uses the elements presented in Figure 6.15

```
//creating objects of Joinpoint.
j1:= new jp1.
j2:= new jp2.
j3:= new jp3.
j4:= new jp4.
j5:= new jp5.
j6:= new jp6.

//creating object of JPDisjunction
jpd1:=JPDisjunction1 new.
jpd2:=JPDisjunction2 new.

//adding the joinpoint objects to the attributes of JPDisjuction.
jpd1.jpset1.add(jp1).
jpd1.jpset1.add(jp2).
jpd1.jpset1.add(jp3).
jpd1.jpset1.add(jp4).

jpd2.jpset2.add(jp5).
jpd2.jpset2.add(jp6).

//creating objects of Advice
ad1:= beforeAdvice1 new.
ad2:= beforeAdvice2 new.

//adding the JPDisjunction object to the attributes of Advice.
ad1.myPointcut1.add(jpd1).
ad2.myPointcut2.add(jpd2).

//creating object of AspectLogger
aspect:=TraceAspect new.

//adding the Advice objects to the attributes of AspectLogger.
aspect.myAdvices.add(ad1).
aspect.myAdvices.add(ad2).

//Aspect installation
aspect.installMe().
```

Chapter 7

CODE GENERATION

XSLT is a declarative, XML-based language used for the transformation of XML[7] documents into other documents, such as XML documents, HTML documents, or plain text documents. The original document is not changed; rather, a new document is created based on the content of an existing one. The XSLT processor takes two input documents - an XML source document, and an XSLT stylesheet—and produces an output document. The XSLT stylesheet contains a collection of template rules: instructions and other directives that guide the processor in the production of the output document. Usually, the query language XPath (The XML Path Language) is used in the XSLT stylesheet for selecting XML document nodes and computing values (e.g., strings, numbers, or Boolean values) from an XML document.

Existing work [12] has demonstrated the use of XSLT (Extensible Stylesheet Language Transformations) for generating XMI[8] to AspectJ code. Because the generic model is compliant with standard UML XMI format and is fully specified in terms of the meta-model, the model to which the profile is applied also becomes compliant with standard UML XMI format. As a result, code can easily be generated. As a

[7] XML [48, 49] is a markup language, which is designed to transport and store data. XML tags are not predefined. It allows the author to define his/her own tags and his/her own document structure [49].
[8] XMI (The XML Metadata Interchange), an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML).

proof-of-concept, we implement two XSLTs which generate valid AspectJ and AspectS code.

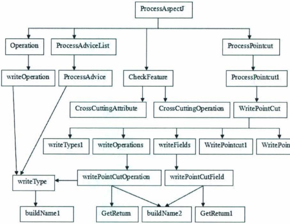


Figure 7.1: Main templates in the XSLT for AspectJ

Code generation for AspectJ is implemented in approximately 1100 lines of XSLT code and consists of 26 templates. On the other hand, code generation for AspectS is implemented in approximately 1250 lines of XSLT code and consists of 31 templates. Some of the templates used in these XSLTs recursively call themselves and thus reduce the lines of code. However, it is possible to make use of more templates and lessen lines of code. The main templates in the AspectJ and AspectS XSLTs are shown in Figure 7.1 and Figure 7.2 respectively. In these diagrams, the arrow sign indicates the call from a template to the other templates. However, some templates recursively call them. These recursive calls to self template are not shown in diagrams.

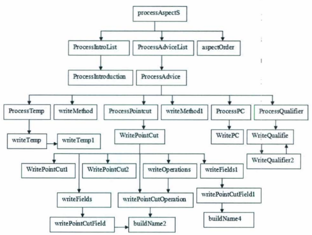


Figure 7.2: Main templates in the XSLT for AspectS

Most of the complexity in the transformation stems from ensuring robustness. The XSLT for AspectJ generates code for packages, aspects, advices and pointcuts. The order of aspects is also followed in the code generation based on the precedence given in a model. Since an args pointcut is generated with each execution pointcut, the modeller must ensure that context exposed from each individual pointcut in a pointcut composition is consistent.

Since the XSLTs are developed as proof of concepts, they are applied to a number of models to show that they produce valid code for those models.

For the AspectJ XSLT, generating code for a composed pointcut at the same time handling the individual pointcuts was the most difficult part. On the other hand, for the XSLT for AspectS, since each advice is related to an advice qualifier attribute that qualifies a pointcut to be advised, the difficult task was tracking and generating code for advices, advice qualifiers and pointcuts that are related. The XSLTs for AspectJ and AspectS are available in the electronic appendices attached to this thesis.

7.1 Application of the XSLT for AspectJ

Existing CASE tools already support code generation for the non-aspect-oriented parts of the model, so that the XSLT only generates code for classes stereotyped as «Aspect» within packages that are stereotyped as «CrossCuttingConcern» [12]. The stereotyped «CrossCuttingConcern» is translated to a package and the stereotyped «Aspect» is translated to an aspect within that package.

For each class stereotyped as «Aspect», using the attribute stereotyped as «AdviceCollection» as reference, the XSLT will generate method stubs for the classes that are stereotyped as «Advice».

A class modeled as a subclass of stereotyped «Joinpoint» is translated to a pointcut. The XSLT also generates method signatures for all the subclasses of stereotyped «Joinpoint». Since AspectJ itself supports only static AOP, using the AspectJ XSLT, the following sections present translation of some static models that were developed in the previous chapter.

7.1.1 ExecutionJoinpoint

In the generic profile (Figure 5.20), we introduced the modeling element ExecutionJoinpoint (Section 5.3.6) to model both call and execution join point selections. A stereotyped «ExecutionJoinpoint» with no senderClass information is translated to an execution pointcut. However, as discussed earlier in (Section 4.1.4.b) .i), a stereotyped «ExecutionJoinpoint» with a senderClass information will be translated to a call() and this() pointcut according to Listing 4.16 and Table 4.5 (second row). Since the args pointcut is implicit in AspectS (Section 4.1.4b) v), for each class stereotyped as «ExecutionJoinpoint», an args pointcut is automatically generated for AspectJ (Section 5.7.10).

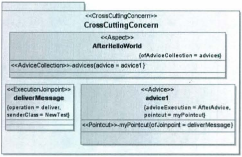


Figure 7.3: An application of the generic profile developed in Section 6.1.2.1

As a proof of concept, the XSLT for AspectJ is used to translate the model related to the senderClassSpecific join point (developed in Section 6.1.2.1) shown in Figure 7.3. Listing 7.1 shows the output of code generation.

Listing 7.1 Code generation for the model shown in Figure 7.3

```
package CrossCuttingConcern;
    aspect AfterHelloWorld {
    pointcut deliverMessage(String message):
            call(public void Test.deliver(..)) && this(NewTest)
                                               && args(message);
    after(String message )  : deliverMessage(message) {}
    }
```

7.1.2 JoinpointDisjunction and JoinpointConjunction

If a class is either stereotyped as «JoinpointDisjunction» or «JoinPointConjuction» the XSLT generates a pointcut that is a composition of multiple pointcuts. The XSLT also generates code for pointcut operators depending on the type of composition. For example, in the shopping-cart example modeled in Section 6.5.2.1 , the classes JPDisjunction1 and JPDisjunction2 were stereotyped as «JoinpointDisjunction». These two composed join points should be translated into two pointcuts that are a composition of multiple pointcuts joined with pointcut operators. The XSLT for AspectJ is used to translate the model shown in Figure 7.4. Listing 7.2 shows the generated code.

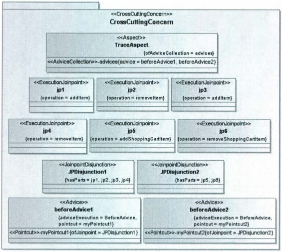


Figure 7.4: An application of the generic profile developed in Section 6.5.2.1

Listing 7.2 Code generation for the model shown in Figure 7.4

```
package CrossCuttingConcern;
aspect TraceAspect {
  pointcut JPDisjunction1(Item item ):
        (jp1(item)||jp2(item)||jp3(item)||jp4(item));
  pointcut jp1(Item item ):
        execution(public void Inventory.addItem(..))
         && args(item);
  pointcut jp2(Item item ):
        execution(public void Inventory.removeItem(..))
         && args(item);
  pointcut jp3(Item item ):
        execution (public void ShoppingCart.addItem(..))
         && args(item);
  pointcut jp4(Item item ):
        execution(public void ShoppingCart.removeItem(..))
         && args(item);

  pointcut JPDisjunction2(
        ShoppingCart sc, Inventory inventory, Item item ):
        (jp5(sc, inventory, item)
```

```
        ||jp6(sc, inventory, item));
    pointcut jp5(
        ShoppingCart sc, Inventory inventory, Item item ):
        execution(
        public void ShoppingCartOperator.addShoppingCartItem(..))
        && args(sc, inventory, item);
    pointcut jp6(
        ShoppingCart sc, Inventory inventory, Item item ):
        execution(
        public void ShoppingCartOperator.removeShoppingCartItem(..))
        && args(sc, inventory, item);

    before(Item item ): JPDisjunction1(item) {}

    before(ShoppingCart sc, Inventory inventory, Item item ):
        JPDisjunction2(sc, inventory, item) {}

}
```

7.1.3 CFlowJoinpoint

In the generic profile (Figure 5.20), we introduced the modeling element CFlowJoinpoint (Section 5.3.6) to model control flow based join point selections that are common for both AspectJ and AspectS.

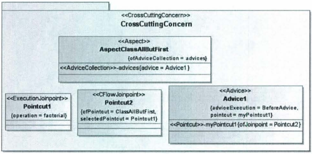


Figure 7.5: An application of the generic profile developed in Section 6.2.2.1

For all the classes stereotyped as «CFlowJoinpoint», the XSLT for AspectJ should generate code based on our discussion in Section 4.1.4 .b) .ii and Table 4.5.

An application of the generic profile on a model related to the cflow join point (developed in Section 6.2.2.1) is shown in Figure 7.5. Following is the translation of that model using the AspectJ XSLT.

Listing 7.3 Code generation for a model shown in Figure 7.5

```
package CrossCuttingConcern;
  aspect AspectClassAllButFirst {

  pointcut Pointcut2(): cflowbelow(execution(
       public long AsFactorialM.factorial(..)))
       && execution(public long AsFactorialM.factorial(..));

  before()  : Pointcut2() {}
}
```

As shown in Listing 7.3, for a stereotyped «CFlowJoinpoint» of type ClassAllButFirst, the XSLT generates a composition of cflowbelow and execution pointcut, which is similar to the emulation shown in Section 4.1.4 .b) .ii. For other types of cflow join points, code generation using the XSLT also complies as discussed in the same section.

7.1.4 ExceptionJoinpoint

In the generic profile (Figure 5.20), the modeling element ExceptionJoinpoint (Section 5.3.7) is introduced to model exception join point selection for AOP languages. For each class stereotyped as «ExceptionJoinpoint», the AspectJ XSLT should generate a pointcut that is composed with an exception handler execution pointcut specifying an

Exception class, an args pointcut, and a cflowbelow pointcut that takes the corresponding execution pointcut as argument.

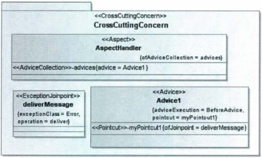


Figure 7.6: An application of the generic profile developed in Section 6.3.2.1

An application of the generic profile on a model related to an exception join point (developed in 6.3.2.1) is shown in Figure 7.6. For that model, code generation, which certainly complies with our above discussion, is shown in Listing 7.4.

Listing 7.4 Code generation for the model shown in Figure 7.6

```
package CrossCuttingConcern;
    aspect AspectHandler {
    pointcut deliverMessage(Error e): handler(Error)
                && args(e)
                && cflow(execution(
                public void TestHandler.deliver(..)));
    before()   : deliverMessage() {}
    }
```

7.1.5 PropertyJoinpoint

In the generic profile (Figure 5.20), we introduced the modeling element PropertyJoinpoint (Section 5.3.8) to model both get and set join point selections. As a proof of concept, an application of the generic profile on a model related to field access join points is shown in Section 6.4 . Each class stereotyped as «GetJoinpoint» should be translated to a get pointcut using the AspectJ XSLT. Similarly, the XSLT should generate a set pointcut for each stereotyped «SetJoinpoint».

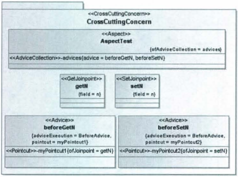


Figure 7.7: An application of the generic profile developed in Section 6.4.2.1

Listing 7.5 presents code generation for the model presented in Figure 7.7. Besides generating the pointcuts, the XSLT also generates appropriate field signatures for both cases.

Listing 7.5 Code generation for the model shown in Figure 7.7

```
package CrossCuttingConcern;
  aspect AspectTest {
     pointcut getN(): get(public int AsCounterModified1.n);
     pointcut setN(): set(public int AsCounterModified1.n);

     before()  : getN() {}
     before()  : setN() {}
}
```

7.2 Application of the XSLT for AspectS

Like the XSLT for AspectJ, the XSLT for AspectS only generates code for classes stereotyped as «Aspect» within packages that are stereotyped as «CrossCuttingConcern» [12]. The stereotyped «CrossCuttingConcern» is translated to a category and the stereotyped «Aspect» is translated to an aspect within that category.

Code for aspect installation and uninstallation will be generated from the information given in the model for both the operations that are stereotyped as «install» and «uninstall» of stereotyped «Aspect». The attribute that is stereotyped as «AdviceCollection», of a stereotyped «Aspect» will be translated to an instance variable of that aspect. The XSLT, for each advice, will generate a method that returns an object of type stereotyped «Advice».

A class modeled as a subclass of stereotyped «Joinpoint» is translated to a pointcut that is a set of AsJoinPointDescriptor objects. The XSLT also generates code for advice qualifier attributes for all the subclasses of stereotyped «Joinpoint». Since AspectS supports dynamic AOP using the AspectS XSLT, the following sections present translation of some dynamic models that were developed in the previous chapter.

7.2.1 ExecutionJoinpoint

A stereotyped «ExecutionJoinpoint» with no senderClass information is translated to a receiverClassSpecific pointcut. From the attribute operation, XSLT gets information about the class and the method whose execution is selected as join point. Based on that information, the XSLT generates code for targetClass and targetSelector.

However, a stereotyped «ExecutionJoinpoint» with a senderClass information will be translated a senderClassSpecific pointcut of AspectS.

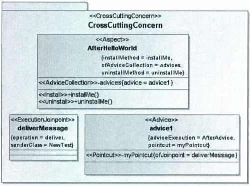


Figure 7.8: An application of the generic profile developed in Section 6.3.2.2

As the context exposing pointcut (similar to args() in AspectJ) is implicit in AspectS, unlike the AspectJ XSLT, the AspectS XSLT does not generate code for the args pointcut with each receiverClassSpecific or senderClassSpecific pointcut.

As a proof of concept, the XSLT for AspectS is used to translate the model related to the senderClassSpecific join point (developed in Section 6.1.2.2) shown in Figure 7.8. Listing 7.6 shows the output of code generation.

Listing 7.6 Code generation for the model shown in Figure 7.8

```
AsAspect subclass: #AfterHelloWorld
     instanceVariableNames: 'advices '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'CrossCuttingConcern '!

     !AfterHelloWorld methodsFor: 'as yet unclassified' stamp:' '!
```

```
adviceAdvice1
    |deliverMessage|
    deliverMessage:=AsJoinPointDescriptor
              targetClass: Test targetSelector: #deliver.

    ^ AsBeforeAfterAdvice
    qualifier:(AsAdviceQualifier
                   attributes: {#senderClassSpecific.})
    pointcut: deliverMessage
    afterBlock:[:receiver :arguments :aspect :client :return| ]! !

    Object subclass: #Main
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CrossCuttingConcern '!

    ! Main methodsFor: 'as yet unclassified' stamp:' '!
    run
    |demoAfterHelloWorld |
    demoAfterHelloWorld :=AfterHelloWorld new.
    demoAfterHelloWorld install.
    demoAfterHelloWorld uninstall.

    ! !
```

7.2.2 JoinpointDisjunction and JoinpointConjunction

The earlier Section 4.1.4c) presented how join point selections that are composed with pointcut operators of AspectJ can be emulated using set operations in AspectS. The dynamic shopping-cart example, which consists of composed join points, was modelled in Section 6.5.2.2 . Using that model and the AspectS XSLT, Listing 7.7 presents code generation for AspectS.

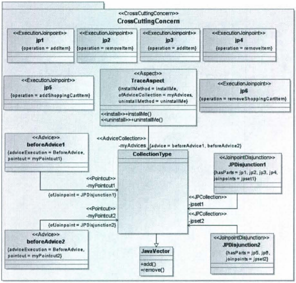


Figure 7.9: An application of the generic profile developed in Section 6.5.2.2

If a class is either stereotyped as «JoinpointDisjunction» or «JoinPointConjuction», the XSLT should generate code as discussed in Section 4.1.4c) . For example, in the dynamic shopping-cart example modeled in Section 6.5.2.2 the classes JPDisjunction1 and JPDisjunction2 were stereotyped as «JoinpointDisjunction». These two composed join points should be translated to two pointcuts that are compositions of multiple pointcuts joined using the function union of set operation. The classes jp1, jp2, jp3, jp4, jp5, and jp6 that are stereotyped as «ExecutionJoinpoint», should be translated as a set of AsJoinpointDescriptor objects. Since the value of adviceExecution is BeforeAdvice for both beforeAdvice1 and beforeAdvice2, they should be translated to two advice objects that consist before blocks. Listing 7.7 shows the translation of the model shown in Figure 7.9 (developed in Section 6.5.2.2).

Listing 7.7 Code generation for the model shown in Figure 7.9

```
AsAspect subclass: #TraceAspect
instanceVariableNames: 'myAdvices '
classVariableNames: ''
poolDictionaries: ''
category: 'CrossCuttingConcern '!

!TraceAspect methodsFor: 'as yet unclassified' stamp:' '!
adviceBeforeAdvice1

|JPDisjunction1 jp1 jp2 jp3 jp4|

JPDisjunction1:=(jp1 union:(jp2 union:(jp3 union:jp4)))
jp1:=AsJoinPointDescriptor
      targetClass: Inventory targetSelector: #addItem.
jp2:=AsJoinPointDescriptor
      targetClass: Inventory targetSelector: #removeItem.
jp3:=AsJoinPointDescriptor
      targetClass: ShoppingCart targetSelector: #addItem.
jp4:=AsJoinPointDescriptor
      targetClass: ShoppingCart targetSelector: #removeItem.

^ AsBeforeAfterAdvice
 qualifier:(AsAdviceQualifier
      attributes: {#receiverClassSpecific.})
 pointcut: JPDisjunction1
 beforeBlock: [:receiver :arguments :aspect :client |      ]! !


!TraceAspect methodsFor: 'as yet unclassified' stamp:' '!
adviceBeforeAdvice2

|JPDisjunction2 jp5 jp6|

JPDisjunction2:=(jp5 union:jp6)
jp5:=AsJoinPointDescriptor
      targetClass: ShoppingCartOperator
      targetSelector: #addShoppingCartItem.
jp6:=AsJoinPointDescriptor
      targetClass: ShoppingCartOperator
      targetSelector: #removeShoppingCartItem.
```

```
^ AsBeforeAfterAdvice
 qualifier:(AsAdviceQualifier
      attributes: {#receiverClassSpecific.})
 pointcut: JPDisjunction2
 beforeBlock: [:receiver :arguments :aspect :client |    ]! !

Object subclass: #Main
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'CrossCuttingConcern '!

! Main methodsFor: 'as yet unclassified' stamp:' '!
run
|demoTraceAspect |
demoTraceAspect :=TraceAspect new.
demoTraceAspect install.
demoTraceAspect uninstall.

! !
```

7.2.3 CFlowJoinpoint

In Section 4.1.4 .b) .ii, we described how some of the control flow based pointcut selections of AspectS can be emulated using the AspectJ constructs. An application of the generic profile on a dynamic model related to cflow join point (classAllButFirst) is shown in Section 6.2.2.2 . According to our design decision a stereotyped «CFlowJoinpoint» that possesses classAllButFirst as the value of its tag cfPointcut will be translated as a ClassAllButFirst pointcut of AspectS. If the model shown in Figure 6.6 is translated using the AspectS XSLT, the value of advice qualifier attribute should be generated as set of two symbols: receiverClassSpecific and classAllButFirst. The aspect, advice and pointcuts should be translated as discussed in previous sections.

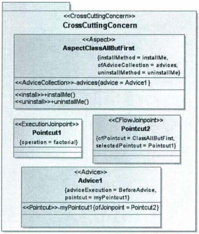


Figure 7.10: An application of the generic profile developed in Section 6.2.2.2

As shown in Listing 7.3, the XSLT generates code for the model shown in Figure 7.10 (developed in Section 6.2.2.2). Similarly, for other types of cflow join points, the AspectS XSLT will be able to generate code.

Listing 7.8 Code generation for the model shown in Figure 7.10

```
    AsAspect subclass: #AspectClassAllButFirst
    instanceVariableNames: 'advices '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CrossCuttingConcern '!

!AspectClassAllButFirst methodsFor: 'as yet unclassified' stamp:''!
    adviceAdvice1

    |Pointcut2|

    Pointcut2:=AsJoinPointDescriptor
            targetClass: AsFactorialM targetSelector: #factorial.
```

```
^ AsBeforeAfterAdvice
 qualifier:(AsAdviceQualifier
   attributes: {#receiverClassSpecific.#cfAllButFirstClass.})
 pointcut: Pointcut2
 beforeBlock: [:receiver :arguments :aspect :client |    ]! !


Object subclass: #Main
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'CrossCuttingConcern '!

! Main methodsFor: 'as yet unclassified' stamp:' '!
run
|demoAspectClassAllButFirst |
demoAspectClassAllButFirst :=AspectClassAllButFirst new.
demoAspectClassAllButFirst install.
demoAspectClassAllButFirst uninstall.

! !
```

7.2.4 ExceptionJoinpoint

In this section, we use the application example related to the ExceptionJoinpoint from Figure 6.9. In the generic profile we introduced the modeling element ExceptionJoinpoint to model the AsHandlerAdvice of AspectS. The class deliver message, which is stereotyped as «ExceptionJoinpoint», should be translated to an AsHandlerAdvice object. Instead of generating a before or after block, the XSLT should generate a handler block. Listing 7.9 shown code generation for the model (developed in Section 6.3.2.2) shown in Figure 7.11.

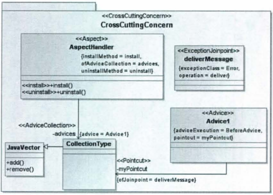


Figure 7.11: An application of the generic profile developed in Section 6.3.2.2

Listing 7.9 Code generation for the model shown in Figure 7.11

```
AsAspect subclass: #AspectHandler
instanceVariableNames: 'advices '
classVariableNames: ''
poolDictionaries: ''
category: 'CrossCuttingConcern '!

!AspectHandler methodsFor: 'as yet unclassified' stamp:' '!
adviceAdvice1
|deliverMessage|
deliverMessage:=AsJoinPointDescriptor
  targetClass: TestHandler targetSelector: #deliver.

^ AsHandlerAdvice
qualifier:(AsAdviceQualifier
   attributes: {#receiverClassSpecific.})
pointcut: deliverMessage
exception: Error
handlerBlock:
[:receiver :arguments :aspect :client :clientMethod |]! !

Object subclass: #Main
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'CrossCuttingConcern '!
```

```
! Main methodsFor: 'as yet unclassified' stamp:' '!
run
|demoAspectHandler |
demoAspectHandler :=AspectHandler new.
demoAspectHandler install.
demoAspectHandler uninstall.
```

7.2.5 PropertyJoinpoint

In Section 4.1.1 .c) , we described the emulation of AspectJ's field access join point selection using AspectS constructs. In the generic profile (Figure 5.20), we introduced the modeling element PropertyJoinpoint (Section 5.3.8) to model both get and set join point selections. As a proof of concept, an application of the generic profile on a dynamic model related to field access join points is shown in 6.4.2.2 . Since AspectS does not have any pointcut constructs to select field access directly, we emulated field access join points of AspectJ by generating and selecting methods that are accessing field as join points in AspectS.

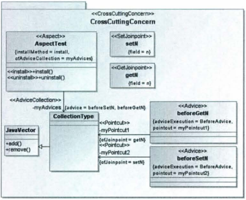


Figure 7.12: An application of the generic profile developed in Section 6.4.2.2

During code translation, each class stereotyped as «GetJoinpoint» should be translated to an AsJoinpointDescriptor Object that specifies the getter method as targetSelector. If a class is stereotyped as «SetJoinpoint», it should be translated to an AsJoinpointDescriptor Object that specifies the setter method as targetSelector. For both cases, the AspectS XSLT should generate code for the related method (getter or setter). Listing 7.10 presents code generation for the model presented in Figure 7.12 (developed in Section 6.4.2.2).

Listing 7.10 Code generation for the model shown in Figure 7.12

```
    AsAspect subclass: #AspectTest
    instanceVariableNames: 'myAdvices '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CrossCuttingConcern '!

!AsCounterModified1 methodsFor: 'as yet unclassified' stamp:' '!
    n:field
    n := field. ! !

    !AspectTest methodsFor: 'as yet unclassified' stamp:' '!

  adviceBeforeSetN
    |setN |
    setN:=[{AsJoinPointDescriptor
            targetClass: AsCounterModified1 targetSelector: #n:.}]

    ^ AsBeforeAfterAdvice
     qualifier:(AsAdviceQualifier
           attributes: {#receiverClassSpecific.})
     pointcut: setN
     beforeBlock: [:receiver :arguments :aspect :client |      ]! !

  !AsCounterModified1 methodsFor: 'as yet unclassified' stamp:' '!
   n
   ^n ! !

   !AspectTest methodsFor: 'as yet unclassified' stamp:' '!

  adviceBeforeGetN
    |getN |
    getN:=[{AsJoinPointDescriptor
              targetClass: AsCounterModified1 targetSelector: #n.}]
```

```
^ AsBeforeAfterAdvice
 qualifier:(AsAdviceQualifier
          attributes: {#receiverClassSpecific.})
 pointcut: getN
 beforeBlock: [:receiver :arguments :aspect :client |    ]! !

Object subclass: #Main
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'CrossCuttingConcern '!

! Main methodsFor: 'as yet unclassified' stamp:' '!
run
|demoAspectTest |
demoAspectTest :=AspectTest new.
demoAspectTest install.
demoAspectTest uninstall.

! !
! !
```

Chapter 8

Discussion

Aspect Oriented Software Development (AOSD) is rooted in the need to deal with requirements that cut across the primary modularization of a software system. On the programming level, we have several AOP implementations for existing programming languages. For example, AspectJ (for Java), AspectC++ (for C++), Aspect# (for C#), AspectS (for SmallTalk or Squeak) and AspectML (for ML) are some of the popular AOP language implementations. However, on the modeling level, there is as yet little support for AOSD. While there has been prior work on extending UML to AOM, most of the extensions expand UML either by introducing new meta-model classes or new notation elements without providing meta-level support. Furthermore, many of the existing AOM approaches are programming language specific and allow modeling on the platform specific model (PSM) level.

Using the extension mechanisms in UML 2.0, [12] presents a meta-model, which is a UML profile for the AspectJ language (Figure 5.1). The profile for AspectJ allows the specification of a platform-specific model (PSM). Since AspectJ follows the static AOP approach, the extension also does not support dynamic AOSD. While AOP language implementations are rapidly maturing, a platform independent model is necessary to increase the reusability of system.

Building on previous work [12], this research presents a core generic meta-model, which is a profile based on the core features of some AOP languages. In order to cover a wide variety of AOP features, we examined AspectJ, AspectS and AspectML. A reader who is not very familiar with AOP may ask the reason to choose the three AOP languages. This is a relevant issue because the choice of languages affects what ultimately ends up in the "core". Adding another language could further restrict the core, while removing one could expand it. Among the languages we studied, AspectJ and AspectS follow the object oriented approach, whereas AspectML follows a functional approach. On the other hand, AspectJ supports static AOP, whereas both AspectS and AspectML support dynamic AOP. By comparing these three languages, we have chosen features that are common among them.

Based on the selected features, we chose the elements to be modeled in the new profile. The previously developed UML extension [12] for static AOP treats aspects as extensions of the Class meta-class, i.e. a stereotyped class. Within that framework, pointcuts are stereotyped structural features and advices are stereotyped behavioral features, typically operations. However, dynamic approaches represent AOSD concepts as first-class modules. For example, join point descriptors (pointcuts), advice, and aspects are all objects in AspectS. Since this approach is not feasible for dynamic AOM our approach differed from the existing work in [12] by providing appropriate extensions.

As a first step to our generic profile, we present a profile which supports only the static part of AspectJ and AspectS. This helps us to discover the modeling elements that are required for the dynamic profile but missing in the current profile. As the second step, a generic but only dynamic profile (does not provide support for static AOSD) is presented. These two profiles clearly show the difference between static and dynamic AOP in the modeling level. We use the above steps and develop the final generic profile that allows existing UML tools to express AOSD models.

The generic profile is based on the core generic features of different AOP languages. Since the languages other than AspectJ have limited pointcut constructs compared to that of AspectJ, several modeling elements from the AspectJ profile were omitted in the generic profile. Moreover, to allow dynamic AOM, some elements were introduced in this profile. Also, for some cases, like modeling a senderClassSpecific pointcut or a control flow based pointcut of AspectS, the profile provides modeling elements that can be directly translated to the corresponding pointcut of AspectS, whereas for AspectJ, code generation follows an emulation scheme. That is why the profile may seem more like an AspectS profile. However, we consider the AOP features of AspectS as a subset of the AOP features of AspectJ. As a result, except for aspect instantiation or dynamic aspect weaving, whatever we can do in AspectS can also be done in AspectJ. Hence, the profile supports modeling of the features from both AspectJ and AspectS.

The developed model ensures modeling support for static or dynamic AOSD from the same profile. One may argue that, the distinction between static and dynamic AOP is really a low-level programming issue, and not one from a modeling perspective. Someone may also think that that it might have been better to focus specifically on static AOP, and compare several static AOP languages, rather than complicate the picture by including dynamic AOP. One of the things that a dynamic AOP implementation offers is the ability to create instances of AOP elements. This is in contrast to the static approach. For example in Chapter 6, we have seen applications of the generic profile on the static (Figure 6.14) and dynamic (Figure 6.15) models of the shopping-cart example. When the profile is applied to both models, they do not differ while modeling the elements such as package, aspect, and join points. However, the differences can be pointed out while modeling instances of aspects, instances of advice collections, instances of join point collections and aspect weaving. In contrast to the static approach, instance creation and aspect weaving are the ability of dynamic AOP implementation. The pseudo code presented in Listing 6.9 represents instances of different objects and aspect weaving of the dynamic model.

Table 8.1 presents a comparison between the AspectJ profile and the generic profile.

Table 8.1: AspectJ Profile VS Generic Profile

AspectJ Profile	Generic Profile
Commonalities	
Requires no special software support	
Supported by UML XMI model interchange facilities	
Allows all aspect-related concepts to be specified in meta-model terms	
Maintains strict separation of base-model and cross-cutting concerns	
Distinction	
Platform specific	Platform independent
Static	Static or Dynamic

Since the generic profile enables the support for modeling of static or dynamic AOP, the profile looks a bit complicated. However, the complexity of the current profile can be minimized by using different profiles for static and dynamic AOP. Another way to make the profile simpler can be to exclude dynamic AOP from the modeling level. In this case, the modeler does not need to care about modeling the dynamic features. Creation of aspect instance and installing or uninstalling the aspect will be handled automatically during code generation. However, that will ignore the capabilities to model an instance of a join point or an instance of an advice that is assigned to a variable. Also, since there will not be any distinction between the static and dynamic AOP in model level, XSLT will generate code for creating aspect instance along with their installation and uninstallation, no matter whether the model is static or dynamic. That is why, although the profile looks complex, we decide to include the elements for modeling of static AOP, besides making the profile fully compatible with dynamic AOM.

To verify the necessity and correctness of the profile, the generic profile is applied to several examples to make sure that it can express both static and dynamic AOSD. However, applying the profile to models can be tedious since it needs aspect-oriented features to be specified for the modeling elements and their relationships explicitly. As a result, the modeller must be aware of the complete base-system model. This can be solved by using pattern based, textual specification. However, the power of pattern specifications is not available in UML. Also, this type of pattern-based specification, while convenient, also opens the door to inadvertent selection of unintended join points. This problem is known as the fragile pointcut problem [45, 46] and is especially problematic when refactoring [47] the base system code, since pattern-based pointcut specifications depend strongly on the specific design of the base system [12]. In this respect, the explicit specification required by the generic profile is safer. Also, the meta-model integration allows easier model checking and verification. Moreover, if patterns were to be specified using textual attributes in the UML model, special tools would be required to resolve such specifications on the model level, e.g. as part of model-level weaving.

One may argue that explicit specification of all aspect-oriented features makes applications almost as complex as if the cross-cutting functionality had been included using non-aspect methods. The use of the generic profile preserves the modularization and encapsulation of cross-cutting concerns; the main advantage of aspect-oriented modeling.

From the model, code generation is accomplished by working from the UML XMI (XML Model Interchange) format, the standard UML serialization. This is one of the standardized mechanisms and is therefore compatible with existing modeling tools. Existing work has demonstrated the use of XSLT (XML Stylesheet Language Transforms) for generating XMI to AspectJ code. Here we leverage that mechanism. As a proof-of concept, we implement two XSLTs that generate valid code for AspectJ and AspectS. Although code translation can also be achieved using some other method; e.g. for Java based UML tools, as [12] has already shown and we also have translated the entire profile using XSLT, we think this is the best way for model transformation into code.

However, the code generation currently relies on the modeler to verify the model. Although we present a number of OCL constraints as part of the model, others must be developed to support validation.

This proposal has some limitations. Each AOP language may have some features that are unique for that language. Since the unique features go beyond the scope of this proposal, a platform specific model cannot be fully accomplished using this profile.

Also, a modeler from a specific AOP language background may find it difficult in modeling some features for which there is no modeling element available within this profile. Moreover, the XSLTs are not tested for complex pointcut composition, which is supported by AspectJ. As a result, for complex pointcut compositions, XSLT may not come up with a correct translation.

The generic profile as well as the models to which the profile is applied look quite complex. As discussed earlier in this chapter, the complexity of the current profile can be minimized either by using different profiles for static and dynamic AOP or by excluding dynamic AOP from the modeling level. When dynamic AOP is ignored in the model level, a modeler does not need to consider the dynamic features while modeling a system. An instance for each aspect will be created automatically during code generation. Similarly, installation and uninstallation of aspect instances will be handled in the course of code generation. Nevertheless, in contrast to current profile, that will ignore the capabilities to model an instance of a join point or an instance of an advice that is assigned to a variable. Also, ignoring dynamic AOP will eliminate distinction between the static and dynamic AOP in model level. As a result, during code generation instance of an aspect will be created and their installation and uninstallation will be done automatically no matter whether the model is static or dynamic. That is why we keep distinction between static AOP and dynamic AOP in model level. However, the complexity of models may reduce usability of the generic profile.

Furthermore, while this profile includes OCL constraints, most current commercial UML modelling tools lack the ability to enforce them. A modeler should follow the given constraints. Such constraints, if enforced, can significantly reduce the complexity of the code generation.

Chapter 9

CONCLUSION

In this research, we developed a platform independent UML based model (PIM), which is a UML profile for the core generic AOP paradigm. We applied this profile to several models to make sure that the profile supports modeling of static or dynamic AOSD. As a proof-of concept, we implemented XSLTs that generate valid code for our target languages (AspectJ, AspectS).

From a theoretical perspective, the strength of this proposal is a complete specification of core AOP features in UML. It is a generic aspect-oriented modeling extension that captures core AOP features in a single meta-model. The core features are chosen by comparing different AOP implementations that vary from each other in their approaches (static or dynamic) and in the diversity of their features. We considered AspectJ as one of the examined languages, because of the maturity of its development and its wide-spread industrial use [1].

The proposal allows all aspect-related concepts to be specified in meta-model terms. Hence, no textual specifications of special keywords are necessary. This means the models can be easily manipulated or verified, without requiring the parsing of keywords or other textual specifications by special tools.

In contrast to the previous works, the proposed profile supports modeling static or dynamic AOP. The previously developed UML extension [12] for static AOP treats aspects as extensions of the Class meta-class, i.e. a stereotyped class. Since this approach is not feasible for dynamic AOM, our approach differs from the existing work in [12] by providing appropriate extensions. To our knowledge, this is the first complete PIM for generic AOP.

From a practitioner's perspective, using the lightweight, meta-model based extension mechanisms of UML 2.0 makes the theoretically important core AOSD meta-model practically useful as a profile. The profile is supported by UML 2.0 compliant modelling tools. The extension requires no special software support and allows aspect modelling to be used within existing, mature software tools. For example, the work described in this paper was developed using the commercially available tool MagicDraw, version 16.0. It contrasts with earlier proposals, which are not all based on profiles and extend UML either by introducing new meta-model classes, or new notation elements, or both [38, 39]. Those proposals cannot be used with available modelling tools and require specific tool support. Although prior work [12] is also defined in terms of meta-model and does not need any special tool support, it provides the specification of a platform-specific model (PSM) and thus differs from our work.

The proposed technique is supported by UML XMI model interchange facilities, the model extension, as well as any models it is applied to, can be exchanged between different MOF (Meta-Object-Facility) compliant UML modeling tools.

The proposed profile offers advantages as it increases the re-usability of the models, cooperation of developers with different language backgrounds, and future-proofing of the software design.

The present work can be extended in multiple directions in future work. First, the generic profile can be extended to include the AOP language features that are unique for different languages, e.g. including the instance specific pointcuts of AspectS. This will expand the modeling capacity of the current profile by covering more AOP features. However, this may confuse some modeler since features specific to a particular language will not be known to a modeler from different background. Also, this will minimize the re-usability of the models and cooperation of developers with different language.

Second, transformations can be developed to transform the platform-independent models ("PIMs") into platform specific models ("PSMs"). For example, in the field of Model-Driven Engineering (MDE), ATL[9] (ATL Transformation Language) provides ways to produce a set of target models from a set of source models. Hence, transformation from PIM to PSM can be done using ATL. This will allow the definition and implementation of the operations on models, and also provide a chain that enables the automated development of a system from its corresponding models. However, having specific languages to represent model transformations requires understanding their foundations, e.g. the semantics, and the structuring mechanisms. In addition, model transformations are required to be stored in repositories so that they can be managed, discovered and reused.

Third, present work can be developed as a plug-in for Eclipse- a multi-language software development environment. But Eclipse is built on the EMF, not the MOF that underlies UML.

[9] ATL(ATL Transformation Language) is a model transformation language and toolkit

Fourth, while some OCL constraints are presented, others can be developed to further ensure the validity of the models. For example, the XSLT should generate **after ... returning** when a return parameter is included in the advice signature, and should generate **after ... throwing** when a raised exception is modeled for the advice.

Finally, usability studies, for example, exploring the impact of various design decisions for this profile, e.g. textual specification of join points versus the present meta-model based specification, need to be conducted. This will allow analyzing the efficiency and performance of the generic profile. Also, conducting the usability study may open the door to choose a better design decision and modify the profile accordingly.

REFERENCES

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

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented Programming," in *Proceedings European Conference on Object-Oriented Programming*, M. Aksit and S. Matsuoka, Eds. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *Proceedings of ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001*. Heidelberg: Springer Verlag, June 2001, pp. 327–354.

[4] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++ : An Aspect-Oriented Extension to the C++ Programming Language," in *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002, pp. 53–60.

[5] M. D. Prasad and B. Chaudhary, "AOP Support for C#," in *AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software*, 2003, pp. 49–53.

[6] R. Hirschfeld, "AspectS - Aspect-Oriented Programming with Squeak," in *NODe '02: Revised Papers from the International Conference NetObjectDays*

on Objects, Components, Architectures, Services, and Applications for a Networked World. London, UK: Springer-Verlag, 2003, pp. 216–232.

[7] "The squeak homepage," http://www.squeak.org/.

[8] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich, "Aspectml: A Polymorphic Aspect-Oriented Functional Programming Language," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–60, May 2008.

[9] R. Hirschfeld, "AspectS home page," http://map.squeak.org/package/e640e9db-2f5f-4890-a142-effebda68748.

[10] O. M. Group, "Unified modeling language," /urlhttp://www.uml.org/.

[11] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc., 2005.

[12] J. Evermann, "A Meta-Level Specification and Profile for AspectJ in UML," in *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*. New York, NY, USA: ACM Press, 2007, pp. 21–27.

[13] R. Miles, *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[14] J. Brant, B. Foote, R. E. Johnson, and D. Roberts, "Wrappers to the Rescue," in *In Proceedings ECOOP '98, volume 1445 of LNCS*. Springer-Verlag, 1998, pp. 396–417.

[15] E. Hilsdale and J. Hugunin, "Advice Weaving in AspectJ," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2004, pp. 26–35.

[16] A. Assaf and J. Noyé, "Dynamic AspectJ," in *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*. New York, NY, USA: ACM, 2008, pp. 1–12.

[17] W. Gilani, F. Scheler, D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat, "Unification of Static and Dynamic AOP for Evolution in Embedded Software Systems," in *Proceedings of the Sixth International Symposium on Software Composition*, M. Lumpe and W. Vanderperren, Eds., vol. 4829. Braga, Portugal: Lecture Notes in Computer Science, 2007, pp. 216–234.

[18] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*. O'Reilly, October 2004.

[19] A. Marot and R. Wuyts, "Composability of Aspects," in *SPLAT '08: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*. New York, NY, USA: ACM, 2008, pp. 1–6.

[20] C. Hofmann, R. Hirschfeld, and J. Eastman, "Flexible Call-by-call Settlement - An Opportunity for Dynamic AOP," in *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, R. E. Filman, M. Haupt, and R. Hirschfeld, Eds., 2005, pp. 19–26.

[21] W. Schroder-Preikschat, D. Lohmann, F. Scheler, W. Gilani, and O. Spinczyk, "Static and Dynamic Weaving in System Software with AspectC++," in *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2006, p. 214.1.

[22] N. Bencomo, G. Blair, G. Coulson, P. Grace, and A. Rashid, "Reflection and Aspects meet again: Runtime Reflective Mechanisms for Dynamic Aspects," in

AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development. New York, NY, USA: ACM, 2005.

[23] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli, "Jac: An Aspect-based Distributed Dynamic Framework," *Softw. Pract. Exper.*, vol. 34, no. 12, pp. 1119–1148, 2004.

[24] J. Baker and W. Hsieh, "Runtime Aspect Weaving Through Metaprogramming," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2002, pp. 86–95.

[25] A. Popovici, T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002, pp. 141–147.

[26] Y. Sato, S. Chiba, and M. Tatsubori, "A Selective, Just-In-Time Aspect Weaver," in *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 189–208.

[27] H. H. P. III, "Smalltalk: A White Paper Overview," www.cs.pdx.edu/~harry/-musings/SmalltalkOverview.html, March 2004.

[28] O. M. Group, "Unified modeling language: Superstructure," August 2005, document formal/05-07-04.

[29] R. A., T. J., and T. M., "Towards Developing Generic Solutions with Aspects," in *Proceedings of the AOM workshop at AOSD, 2004*, 2004.

[30] R. Pawlak, L. Duchien, G. Florin, F. Legond-aubry, L. Seinturier, and L. Martelli, "A UML Notation for Aspect-Oriented Software Design," in *in Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*, 2002.

[31] M. M. Kande, J. Kienzle, and A. Strohmeier, "From AOP to UML- A Bottom-Up Approach."

[32] M. Basch and A. Sanchez, "Incorporating Aspects into the UML," 2003.

[33] L. Fuentes and P. Sanchez, "Elaborating UML 2.0 Profiles for AO Design," in *Proceedings of the AOM workshop at AOSD, 2006*, 2006.

[34] O. Aldawud, T. Elrad, and A. Bader, "A UML Profile for Aspect Oriented Modeling," in *Proceedings of OOPSLA 2001*, 2001.

[35] ——, "UML Profile for Aspect-Oriented Software Development," in *The Third International Workshop on Aspect Oriented Modeling*, 2003.

[36] D. Stein, S. Hanenberg, and R. Unland, "Designing Aspect-Oriented Crosscutting in UML," in *In AOSD-UML Workshop at AOSD '02*, 2002.

[37] E. Barra, G. Genova, and J. Llorens, "An Approach to Aspect Modelling with UML 2.0," in *Proceedings of the AOM workshop at AOSD, 2004*, 2004.

[38] J. Grundy and R. Patel, "Developing Software Components with the UML, Enterprise Java Beans and Aspects," in *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2001, p. 127.

[39] Y. Han, G. Kniesel, and A. B. Cremers, "A Meta Model and Modeling Notation for Aspectj," in *Proceedings of the AOM workshop at AOSD, 2004*, 2004.

[40] W. Harrison, P. Tarr, and H. Ossher, "A Position on Considerations in UML Design of Aspects," in *Proceedings of the AOM with UML workshop at AOSD, 2002*, 2002.

[41] F. Mostefaoui and J. Vachon, "Formalization of An Aspect-Oriented Modeling Approach," in *Proceedings of Formal Methods 2006*, Hamilton, ON, 2006.

[42] D. Stein, S. Hanenberg, and R. Unland, "An UML-based Aspect-Oriented Design Notation for AspectJ," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002, pp. 106–112.

[43] M. Mosconi, A. Charfi, J. Svacina, and J. Wloka, "Applying and Evaluating AOM for Platform Independent Behavioral UML Models," in *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*. New York, NY, USA: ACM, 2008, pp. 19–24.

[44] J. U. Júnior, V. V. Camargo, and C. V. F. Chavez, "UML-AOF: A Profile for Modeling Aspect-Oriented Frameworks," in *AOM '09: Proceedings of the 13th workshop on Aspect-oriented modeling*. New York, NY, USA: ACM, 2009, pp. 1–6.

[45] K. Gybels and J. Brichau, "Arranging Language Features for More Robust Pattern-based Crosscuts," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2003, pp. 60–69.

[46] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *In Proceedings of the*

European Conference on Object-Oriented Programming (ECOOP. Springer-Verlag, 2006, pp. 501–525.

[47] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[48] D. Hunter, A. Watt, J. Rafter, J. Duckett, D. Ayers, N. Chase, J. Fawcett, T. Gaven, and B. Patterson, *Beginning XML*, 3rd Edition, Ed. Wiley Publishing, Inc., January 2005.

[49] w3schools.com, "Introduction to XML," http://www.w3schools.com/xml/-xml_whatis.asp.