ENABLING EXPERIMENTATION OF ASPECT-ORIENTED PROGRAMMING LANGUAGES THROUGH A META-WEAVER FRAMEWORK

By

MELISSA ANN STEFIK

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2008

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of MELISSA ANN STEFIK find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

This research project would not have been possible without the support of many people. I would like to thank my adviser, Roger Alexander, for his support and guidance throughout this process. I would also like to thank the members of my committee: Robert Patterson and Shira Broschat.

ENABLING EXPERIMENTATION OF ASPECT-ORIENTED PROGRAMMING LANGUAGES THROUGH A META-WEAVER FRAMEWORK

Abstract

by Melissa Ann Stefik, M.S.
Washington State University
May 2008

Chair: Roger Alexander

Modern software engineers deal with software systems of enormous complexity. Large corporations hire teams of programmers, who, at least in theory, cooperate to work on programming projects that can involve millions of lines of computer code. Object-oriented programming was designed to help encapsulate these programs, essentially breaking what was once millions of lines into manageable components that can be understood by an individual. Unfortunately, some of these components, by the shear nature of certain programmatic solutions, are related to many parts of a total software system. These components are said to be global (or semi-global) in nature put another way, they are crosscutting concerns.

Enter aspect-oriented programming, a new paradigm for managing and encapsulating these, so called, crosscutting concerns. In aspect-oriented programming, programmers have commands, built into the language, that allow them to modify pieces of the code on a global scale. Modern AOP languages, however, tend to be language specific and difficult to modify, in part because many languages have custom rules, complex procedures, and a bizarre syntax. A general system for encapsulating these rules could help alleviate the complexity of designing AOP languages and help enable experimentation on this new class of programming languages.

As such, I present CAL, the **C**ustomizable **A**spect **L**anguage, a new system, termed a meta-weaver, for creating, changing, modifying, and experimenting with aspect-oriented programming.

Unlike traditional aspect-oriented languages, like AspectJ, CAL represents aspects, and even the underlying aspect language, as XML. This allows users of CAL to modify the crosscutting concerns in whatever programming language they are writing in, and also allows CAL to crosscut itself, essentially modifying its own global nature. This simplifies the process of changing and adjusting an aspect-oriented programming language and its corresponding weaving rules.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

Dedication

To my husband.

CHAPTER 1

INTRODUCTION

1.1 Thesis Summary

In object-oriented programming, components of a computer system are modularized according to their primary behaviors; for example a dog can walk, a car can be driven, and a fish can swim. We reuse code amongst these components by using inheritance and polymorphism, yet some pieces of this code, often called *cross-cutting*, cannot be easily reused in an object-oriented system. This is caused by *scattering* and *tangling*, where scattering occurs when a single requirement is spread across several modules and tangling occurs when multiple requirements are intertwined with the core responsibility of a single module [5].

Aspect-oriented programming [23], on the other hand, is a new paradigm for modularizing the, so called, *cross-cutting concerns* of a software system. Concerns, which are conceptual units that can include "features, nonfunctional requirements, and design idioms" [32], are called cross-cutting because the code that implements them is often scattered across a set of classes, modules, or interfaces. A typical example is logging, where the developer might want to write code that records the invocation of every method, but would prefer not to literally type the logging statements into every method on the system. In aspect-oriented programming, developers accomplish this in a textually localized manner, which means that scattering only occurs when a program is compiled or executed, a process which is analogous to a sophisticated preprocessor.

In this thesis, I make a customizable, modular, aspect-oriented *weaver*, which I call a *meta-weaver*. A meta-weaver allows the customization of weaving rules, precedence rules, aspect syntax, and aspect semantics. The current state of the art is to use custom weavers for every application or language, yet a meta-weaver allows us to customize the weaving process, which enables experimentation and research on aspect-oriented languages.

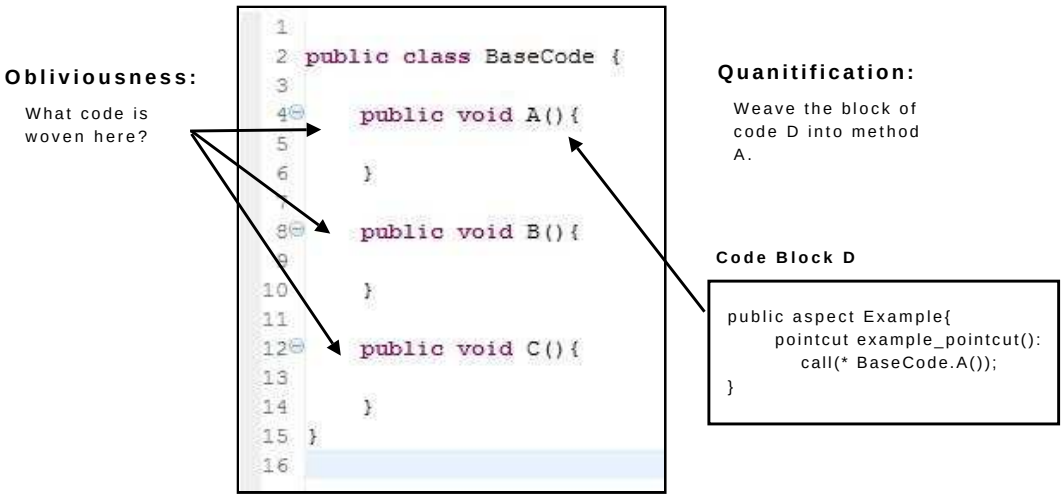

Figure 1.1: Visual example of obliviousness and quantification.

1.2 An AOP Overview

Aspect-oriented programming is a technique for modularizing cross-cutting concerns in a software system. AOP systems are typically considered to have two characteristics that distinguish them from object-oriented programming: obliviousness and quantification [9].

Obliviousness is the idea that an aspect is invisible to the programmer when looking at the source code. This means the programmer cannot determine if or where an aspect will be executed. For example, in Figure 1.1, a programmer is unaware of the aspect and need not be concerned about the function of the aspect. To implement the functionality of the base code, the programmer, ideally, does not need to know anything about the aspect (although this is debatable in practice). In other words, there is no way for this programmer to tell if the aspect adversely affects the base code, or even if an aspect will be applied, without the source for the aspect.

Quantification is the process of connecting the aspect to the base program through statements that identify where the aspect will be integrated. In Figure 1.1, quantification is demonstrated by code block D. Here the aspect identifies method A as a location where the code will be woven and methods B and C are where it will not.

Obliviousness and quantification are implemented through a process called *weaving*. Weaving

integrates aspects into the base program code by identifying locations in the base program. After a programmer specifies what points in the software will be modified by the aspects, the weaver handles actually integrating the aspect code with the base code. This process can be complicated, especially when two aspects indicate that they apply to the same location, which then requires the weaver to determine how the aspects should be integrated and in what order. Traditional weavers, like the one included in AspectJ, arguably the most common aspect-oriented programming language, are not customizable; this makes experimentation on how the weaver should work difficult.

1.3 Challenges of AOP

While aspect-oriented programming has been in existence for many years, there still exists several significant technical challenges, including: ambiguous aspect orderings and legal weaving locations (join points). Aspect orderings often become a problem when different aspects have the same weaving location. Suppose two aspects identify the same location to weave into; in other words, suppose two aspects overlap. Ultimately, the AOP weaver must decide which of these two aspects are integrated into the base code first and which are integrated second. How does the AOP system decide which aspect to weave first?

Overlapping aspects, because the they can have a variety of orders to weave, produce different results in the base program. This makes the process of weaving ambiguous and requires a programmer to specify what the desired outcome is for each overlapped aspect.

Another common problem in aspect-oriented programming is clearly identifying weaving locations. Without a clear way to identify the result of the weaving process, a programmer can identify a set of locations which may or may not include the locations they intended. Worse, there is often no visual way to check if an aspect integrated into the base code or where it was integrated.

Modern AOP languages weave aspects into bytecode representations of base code. While this practice has largely been adopted for efficiency reasons, it has one significant and negative side effect: visibility. When a programmer uses a programming language like AspectJ, after weaving

has occurred, the programmer can literally not see the locations where code was inserted from an aspect. Thus, the weaving locations are often ambiguous, leaving the programmer with little indication of whether he or she achieved the desired results.

To address these problems a meta-weaver can be used to investigate and identify properties of different types of weaving. A meta-weaver accomplishes this by showing the outcome of the weave and allowing the properties of aspects and weaving to be modified. A researcher using a meta-weaver can then identify problems with the weaving process and explore alternative solutions that further clarify the weaving process.

1.4 Problem Statement

Modern AOP languages are not customizable to the extent that easy testing of AOP features is a possibility. With a broad number of aspect languages, each having its own model for aspect orientation, there is a need for a framework that is flexible enough to allow each aspect feature to be quickly changed. I have identified two primary problems which I address in this thesis, namely: "How do we enable experimentation of aspect-oriented languages?" and "How do we easily change the syntax and semantics of aspect-oriented languages?"

1.4.1 How do we enable experimentation of aspect-oriented languages?

Aspect-oriented programming has the relatively unusual attribute of having multitudes of papers written on the topic, yet almost no formal experimentation testing language features. I argue that this is due, in part, because designing new aspect languages and features is notoriously difficult, often requiring expertise in compiler design to even begin the work, as illustrated in Figure 1.2. In other words, to promote experimentation, we need to *enable* a researcher's ability to customize AOP languages with minimal effort. For example, researchers should not have to write a custom compiler to experiment with AOP features.

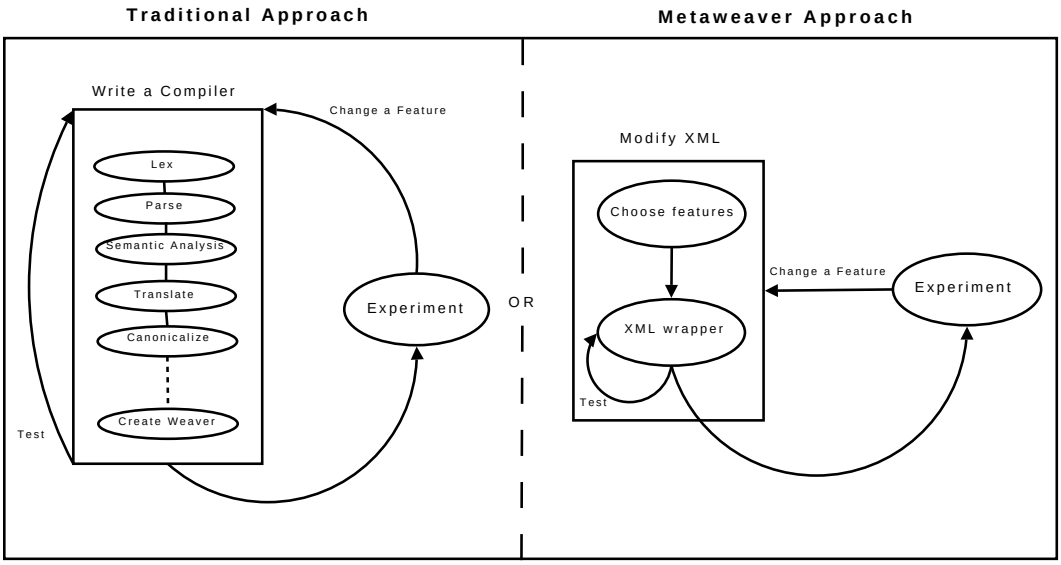

Figure 1.2: Traditional and meta-weaver approaches to designing customizable AOP languages.

1.4.2 How do we easily change the syntax and semantics of aspect-oriented languages?

In a language like AspectJ, altering syntax, semantics, or weaving rules, requires the programmer to write and edit a custom compiler. For example, in Figure 1.2, the traditional approach to adjusting aspect-oriented language features requires the researcher to go through the traditional compiler steps and to develop a weaver as part of the backend of their research. If modifications to the compiler are needed, the researcher may have to modify any number of the traditional compiler phases.

1.5 The Thesis

I argue that because AOP languages are not easily customizable, developing a framework that allows syntax, semantics, and weaving rules to be easily changed will aid in discovering the impact of AOP language features. To demonstrate this, I present CAL, the **C**ustomizable **A**spect **L**anguage, which is an XML based system for creating, changing, modifying, and experimenting with aspect-oriented programming.

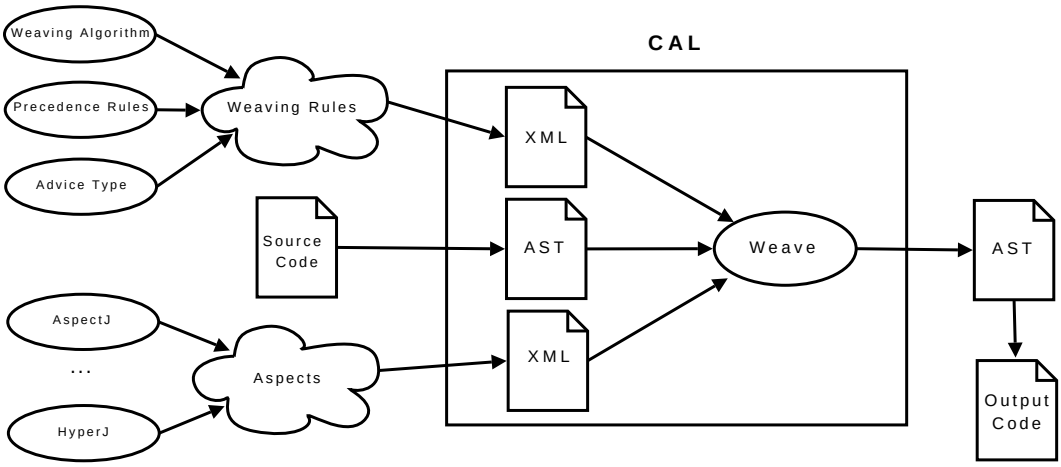

Figure 1.3: General model of the meta-weaver.

1.5.1 Design of the Meta-Weaver

The goal of CAL is to allow experimentation with weaving rules by creating a highly flexible aspect-oriented weaving framework. CAL uses XML to represent features of aspect-oriented languages. Representing these features through XML makes it relatively easy to change the features of a language or modify the way in which an AOP language functions.

Figure 1.3 shows the basic design of CAL. The two clouds in Figure 1.3 represent conglomerates of rules for weaving aspect-oriented code. On the top, *weaving rules* includes features of weaving, including, what type of algorithm accomplishes the weaving, precedence rules, or advice types, while the bottom represents the syntax for aspects. These two clouds are represented in CAL as XML documents, which are used to build intermediate representations of an aspect's syntax and semantics. CAL then takes the XML based weaving rules and uses them to combine base code with advice. The output of this process is a woven abstract syntax tree that is translated into source code.

1.5.2 Applications of the Meta-Weaver

We know relatively little about the effect of the AOP paradigm on programming. As such, a modular AOP system is needed to promote the fine grain testing of various aspect features. The

scope of this testing includes human studies on comprehension of the effects of aspect orientation and the individual impact of syntax, semantics, or even technical concerns related to algorithm efficiency.

1.5.3 Limitations

While CAL is a useful tool for implementing features of an AOP language, and in theory, is also customizable for any arbitrary AOP language, in practice, this type of tool is only as general purpose as the intermediate representation tree it is built upon. In other words, some IR trees are created with a certain language in mind, which may not map completely with other languages. For example, an intermediate representation of Java would only need to account for statically typed variables. However, an intermediate representation for Visual Basic would need to account for dynamically typed variables. Similarly, in CAL, certain aspect-oriented features may need to be added to the XML representation and the weaver.

These same sort of issues are not only expected but anticipated. CAL has an architecture designed, with these limitations in mind; it allows for easy modification and expansion of the base code representations, aspect representations, and weaving rule representations.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we will discuss various characteristics of aspect-oriented programming. The initial focus of this chapter will be, the basic concepts of programming with aspects in, what is likely the most popular language for aspect-oriented programming, AspectJ. I will then discuss how aspect-oriented programming influences or changes the concept of modularity. Finally, I will discuss previous work on both weaving for, and the applications of, AOP.

2.1 Introduction to Aspect-Oriented Programming

Kiczales et al. created aspect-oriented programming [21], a new paradigm for modularizing computer programming. AOP allows us to codify, and more easily manipulate, cross-cutting concerns.

Cross-cutting concerns are aspects of a program which are *scattered* or *tangled* with other modules in a program. Due to the nature of existing programming languages, textually separating cross-cutting concerns from modules can be difficult. Aspect-oriented programming is a tool that allows us to untangle cross-cutting concerns, through the use of *aspects*. Aspects better enforce modularity and are easier to manage from the perspective of the programmer. However, aspects must be redistributed into the base program, in a process called *aspect weaving*, in order to execute [23].

To better understand the concept of a cross-cutting concern we must first understand what a *concern* is. A concern is loosely defined as a design decision of a system that is incorporated into code. Similar to cross-cutting concerns, one of the main concepts is the idea of *separation of concerns* [8, 30], which ideally result in a single design decision in one unit, or module. However, not all concerns can be effectively separated, nor are they easily identified [31]. To separate these design decisions, aspect-oriented programming introduces a unit called an *aspect*. There are several main components that make up an aspect, including *join points*, *pointcuts*, and *advice*.

```
public class Employee
{
   ...
   public double pay(int hoursWorked, double payRate)
   {
       double payment = payRate * hoursWorked;
       return payment;
   }
}
```

Figure 2.1: Part of a program containing join points.

```
execution(double Employee.pay(int,double))
```

Figure 2.2: A pointcut in AspectJ

"A ***join point*** is an identifiable point in the execution of a program" [25]. Join point locations include places in code like assignment statements, method calls, or initializations [25]. This is demonstrated by the code block in Figure 2.1. From this code example, the join points that can be identified are the assignment to the variable `payment` and the call of the method `pay()`.

We can specify a set of join points, like those identified in Figure 2.1, by defining the concept of a ***pointcut***. Pointcuts are specifications in an aspect used to select a set of join points from base code for weaving. Figure 2.2 is an example of a pointcut in AspectJ. This particular pointcut specifies that code should be woven into the method `pay()`.

In addition to join points and pointcuts, an aspect is made up of advice. ***Advice*** is code to be executed at when a join point event, which is specified by a pointcut, occures at runtime. Figure 2.3 shows advice that will output a line of text once execution reaches the join point, which is specified by the pointcut in Figure 2.2. This results in the advice being integrated into the base program after the execution of the method `pay()`. In this example, the print statement is the advice block or code to be executed. The command `after()` specifies that the advice will be woven after the join point.

Figure 2.4 shows join points, pointcuts, and advice, together as a complete ***aspect***. This is

```
after() : execution(double Employee.pay(int,double)){
    System.out.println(‘‘Made payment to employee.’’);
}
```

Figure 2.3: An example of advice in AspectJ

similar to the declaration of a class, which contains any number of methods. Additionally, an aspect can contain multiple pieces of advice and any number of pointcuts.

In addition, Figure 2.4 shows a variation to the poincut given in Figure 2.2. This new pointcut declaration is broken into several parts, the *access specifier*, *keyword*, *pointcut name*, *pointcut type*, and *signature*. In AspectJ, there are several pointcut types, including *call*, and *execution*. In this example we use the type *call* to specify that the pointcut will occur when the method `pay()` is called. This pointcut also specifies join points for any method named pay, regardless of the type and number of its arguments. This is accomplished through the use of dots in the parameter list.

Join points, pointcuts, and advice are tightly linked through the process of ***weaving***. Weaving is the process of connecting aspects with the base program, where the base program is the original program written in a programming language, such as, Java or C++, and the aspects are written in a corresponding aspect-oriented language, AspectJ or AspectC++. Weaving in AspectJ will be discussed later in this chapter.

2.2 Modularity

One of the key concepts involved in aspect-oriented programming, and programming in general, is modularity. Modularity is a property of a computer program and is a measure of the extent to which code is separated into modules. Ideally, each module captures a single design decision. Modularity has several key characteristics, including the idea that modules are *textually local*, that each module has a well defined interface, and that this interface is an abstraction of its implementation [24].

Though the concept of modularity applies to both aspect-oriented programming and object-oriented programming, object-oriented programming modularizes program entities or concerns

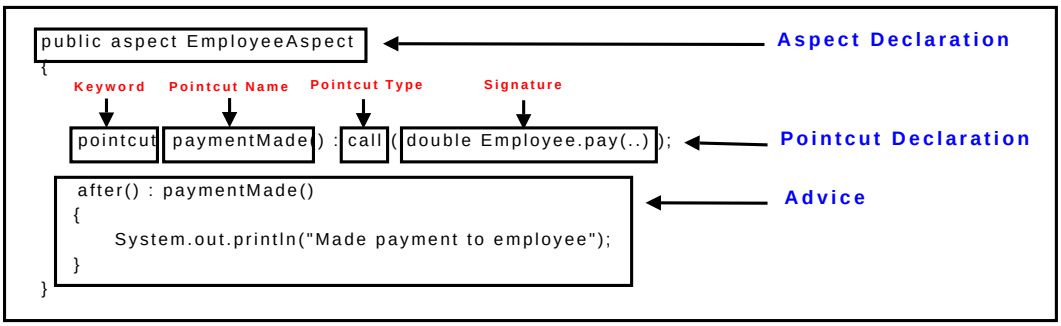

Figure 2.4: An example of an aspect in AspectJ.

and aspect-oriented programming modularizes cross-cutting concerns. The extent to which aspect-oriented programming promotes modularity of cross-cutting concerns has been well explored [1, 7, 6]. Sullivan et al. [37] and Lopes [28] further explore this issue using a Design Structure Matrix to evaluate the modularity of a program. This is accomplished by modeling the design of a program and then comparing it to other models. Lopes applies this idea to aspect-oriented programming and shows aspects can be treated as modules. In addition, Lopes demonstrates that AOP does increase modularity in certain cases, including, when an existing system needs to be augmented.

Although Steimann casts some doubt as to whether modularity is really increased when using aspect-oriented programming, his main focus is on the claim that, "AOP has set out to modularize crosscutting concerns (its methodological claim), but by its very nature (its mechanics) breaks modularity..." [35]. He argues that AOP does not increase understandability and can often stand in the way of independent development, one of the fundamental reasons for modularity. In addition, Steimann argues that most of the proposed or demonstrated applications of AOP are not necessarily good examples. As Steimann points out, logging, tracing, and debugging are the canonical examples of uses for AOP, but they are also applications that are already well established and at best AOP is an alternative solution to these problems.

Aldrich presents the idea of an Open Module. An open module allows much of the same functionality of existing AOP but it preserves modularity and information hiding by sacrificing obliviousness [1]. This is accomplished by allowing a module to export its own pointcuts, which

are abstractions of the cross-cutting concerns within the module, making them available to advice [2]. There has been some debate on whether Open Modules are cross-cutting because of the restrictions between the modules and advice. Sullivan et al. propose an alternative solution that creates a design rule interface between aspects and advice code which addresses these issues [36].

Finally, Kiczales and Mezini propose another way of thinking about modularity with respect to aspect-oriented programs, modular reasoning [24]. *Modular reasoning* allows decisions to be made about a module with access to only that module and the interfaces which are directly referenced in that module. They argue that modular reasoning is achieved through aspect-oriented programming, though, it requires some global knowledge of the system to be obtained. Kiczales and Mezini explain that the presence of cross-cutting concerns, with or without aspect-oriented programming, requires global knowledge. Thus, AOP does not itself break modularity, but cross-cutting concerns by their nature create the need for global knowledge, which does break modularity. They claim that AOP simply provides the benefit of modular reasoning once global knowledge is identified.

2.3 Weaving

Weaving is the process of integrating advice into the execution of a program. An integral part of this is identifying join points, pointcuts, and advice. While weaving might seem straight forward, there are many approaches to the weaving process, each depending on the language, the aspect-oriented language extension, and the weaving rules. In this section, some of these approaches to weaving will be discussed.

Hilsdale and Hugunin discuss the AspectJ weaving process as it existed in AspectJ 1.1 [19]. In AspectJ, weaving is achieved by matching join points to pointcuts through *join point shadows*. Join point shadows are locations in bytecode or source code that represent the actual join points at runtime. This process of weaving first requires a bytecode transformation; then the join point shadows must be matched to join points and advice. They also discuss the efficiency of the weaving process and explain that weaving into bytecode is more efficient than weaving into source code.

Most aspect-oriented languages are designed to work with a corresponding *base language* like AspectJ and Java. This is prohibitive because it requires an aspect language to be developed for each language. To address this issue, the concept of the *meta-weaver* was introduced by Gray to allow weavers to be created through meta-specifications. These meta-specifications provide all the necessary details of a base and aspect language to generate a weaver [12]. Ideally, a weaver could be easily generated or would be flexible enough to be language independent. To accomplish this, Gray lays out a framework for the meta-weaver [14]. This framework was eventually developed and is able to generate aspect weavers for different languages [15]. However, each new weaver requires a certain amount of customization and extra effort to program.

Lafferty and Cahill, present Weave.NET, which is a language independent weaver that allows aspects to be written in a variety of languages [26]. They accomplish this through the use of XML to specify aspects, which prevents aspects from being intertwined with a particular language. Weave.NET, however, uses the AspectJ notation for aspects; it is not flexible enough to allow experimentation with different weaving rules because it is limited to AspectJ notation and weaving into bytecode.

While Gray, Lafferty, and Cahill all provide methods similar to the work presented here, CAL allows the user to build up and modify a complex set of weaving rules and language features. In CAL, an XML specification allows users to build up a set of weaving rules, while Gray creates a top-down framework which generates weavers, but does not allow a user to necessarily build specific weaving rules. One could argue that Gray's work is at a higher level of abstraction than mine, specifically in relation to model composition and domain specific languages. CAL, in contrast, deals with building up a detailed and flexible framework for weaving different types of aspects.

Similarly, Lafferty and Cahill's work, as the name weave.NET implies, works only for Microsoft based languages, and since it uses the syntax of AspectJ, it is fundamentally limited to the semantics of AspectJ. In contrast, CAL allows the user to specify not only the weaving rules of AspectJ, but rules used by other AOP languages. In addition, CAL allows the user to customize

the join points, pointcuts, and precedence rules available as part of the core language, making CAL considerably more flexible than weave.NET.

2.4 Applications of AOP

Let us now consider some of the practical applications of aspect-oriented programming. In this section, we discuss several case studies of how AOP has been applied in areas such as design patterns, web based programming, and operating systems. This work is related to my own (i.e., CAL) in the sense that CAL provides a framework for experimenting with AOP languages, facilitating studies such as those presented in this section.

One such application is the Gang-of-Four (GoF) design patterns, which Hannemann and Kiczales integrate with AOP [18]. Specifically, they conduct a study which compares the implementation of 23 GoF design patterns in Java to their implementation in AspectJ. The authors' goals were to examine and compare the modularity and reuse of the GoF design patterns in the previously mentioned languages. They reported 17 of 23 design patterns in AspectJ were more modular than their counter parts in Java. In addition, the authors claim 12 of those implementations increased reuse.

Further, Garcia et al. conducted a quantitative study on aspect-oriented and object-oriented implementations of those same 23 GoF design patterns [11]. The study confirmed an increase in separation of concerns. However, as Garcia points out, "separation of concerns can not be taken as the only factor to conclude for the use of aspects" [11]. In addition, they found some of the aspect-oriented solutions have more complex operations and coupled components. In a later study, Cacho et al. expand further on these ideas by investigating the scalability of AOP on GoF design patterns [4].

Beyond design patterns, web-based applications of AOP have also been explored. Kersten and Murphy ran one of the earliest case studies using AOP and AspectJ to build a web-based learning program, Atlas [20]. Using AspectJ, the authors built an advanced teaching and learning academic

server, for which they documented what aspects were used and how those aspects affected the development process. From this they were able to identify possible hurdles a programmer might face when using aspect-oriented programming.

In addition, Papapetrou and Papadopoulos present a case study comparing aspect-oriented and object-oriented programming on a component-based web-crawling system [29]. They used several metrics to measure effectiveness, learning curve, time to complete, code tangling, and stability of the resulting software. They also narrowed the focus of the study to three specific areas of their program: logging, overloading checks, and a database optimizer. Papapetrou and Papadopoulos report relatively positive results for the aspect-oriented solution. This includes a shorter implementation time and less code tangling, especially in the case of logging. However, both the aspect-oriented and object-oriented solutions resulted in stable and effective solutions.

Lohmann et al. explored aspects used in the development of operating system kernels [27]. For the eCos kernel the authors used AspectC++ in the implementation of this operating system kernel which requires a highly efficient infrastructure. The authors discovered most aspect-oriented features do not introduce extra runtime overhead. However, they identified some areas of aspect-oriented programming that can cause extra overhead such as ambiguous join points.

A framework such as CAL facilitates exploration of these applications of aspect-oriented programing. For example, changes to certain language features in AspectJ could improve modularity and reuse of the GoF design patterns or help mitigate common problems a programmer faces when using an AOP language. The current state-of-the-art, however, makes exploring these applications difficult, as modifying a language like AspectJ takes considerable technical savvy and effort. With CAL, however, the technical requirements are mitigated, making exploring the applications of AOP languages easier.

CHAPTER 3

CUSTOMIZABLE ASPECT LANGUAGE (CAL)

The goal of CAL is to provide a framework for modifying both the features of an aspect-oriented language and a framework to modify the way transformations of aspects occur. This is similar to Gray's meta-weaver in that a side effect of this work is language independence. Although Gray's meta-weaver is a general framework that generates domain specific weavers which are specific to a base language (also called the target language) [13]. CAL extends these capabilities by providing modifiable weaving rules which allow more control over the weaving process and the resulting program after weaving.

CAL provides a framework which allows the development of weavers for various target languages and weaving rules. In this chapter, the specifics of this framework will be discussed, first by outlining the overall design of the meta-weaver and then presenting detailed descriptions of each of the components that make up this meta-weaver. The representation of aspects in XML will be used as an example of how CAL is general enough to modify and handle a variety of aspect-oriented languages. Finally, I will discuss the implementation of the components of the meta-weaver and how its design enables modifications to the weaving process.

3.1 CAL Overview

There are three primary components in CAL, the input, the meta-weaver, and the output. Figure 3.1 gives an overview of these components and the design of CAL. Using several examples, in this section I will discuss the design of each component and the reasoning behind their design. Finally, I will discuss how the input and the meta-weaver work together to produce a program that combines the base and aspect programs.

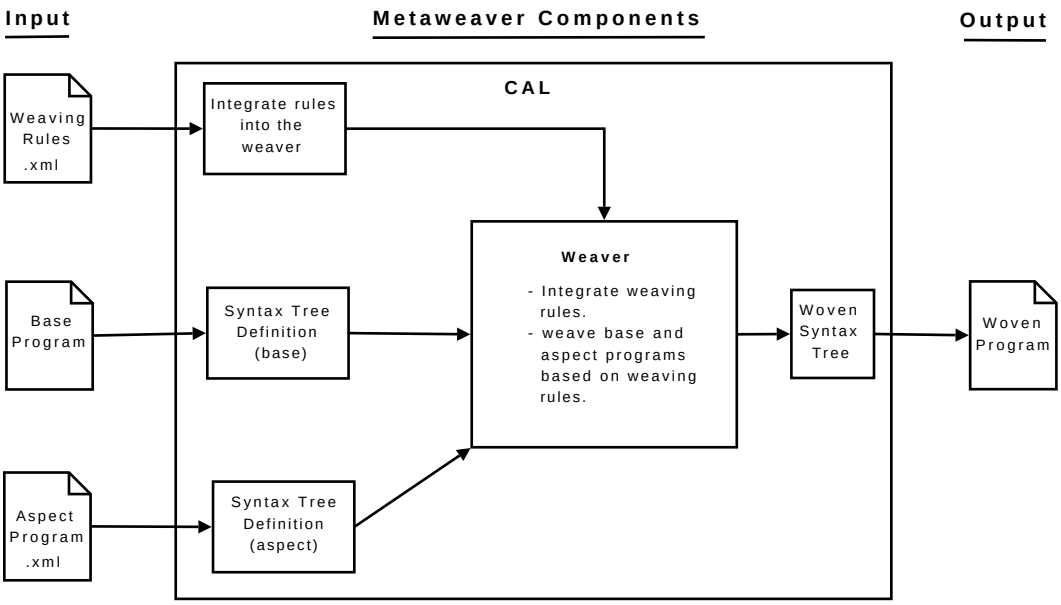

Figure 3.1: The CAL Framework.

3.1.1 CAL Input

CAL has three inputs: the weaving rules, the base program, and the aspect program. Each of these inputs has its own format that contributes to providing the meta-weaver with the necessary information to modify the weaving process and produce a combined output program.

The first type of input is an XML file which is specified by the schema shown in Appendix B.2. This XML file contains weaving rules. These weaving rules specify what types of advice are legal and the order in which this advice is integrated into a base program. The meta-weaver then acts as an interpreter of these rules. In essence, this allows a user to modify or add rules that will change the way the weaver combines the base program with the aspect program.

The second type of input is the base program. Figure 3.2 is a simple example of a possible base program written in Java. This particular program creates an *Account*, deposits an amount of $100.00 into that *Account*, and displays the current balance. While the input is Java, it is possible to input a base program in another language. The input is only limited by the abstract syntax tree representation. Modification and revision of the abstract syntax tree allows CAL to accept base

```
public class Account {
    private double balance;
    public void deposit(double amount){
        balance += amount;
    }
    public void printBalance(){
        System.out.println("Balance: " + balance);
    }
}

-------------------------------------------------

public class Main {
   public static void main(String[] args) {
      double amount = 100.00;
      Account account = new Account();
      account.deposit(amount);
      account.printBalance();
   }
}
```

Figure 3.2: Example base program written in Java.

programs in any language.

The final type of input is an XML file representing an aspect. Aspects are typically written in an aspect-oriented language, for example AspectJ, that extends a base language like Java. In contrast, CAL represents aspects through the use of XML, as demonstrated by the schema shown in Appendix B.1. XML allows a developer to extend functionality and combine different approaches to aspect-orientation (e.g., weaving rules, join point definitions, precedence rules). Figure 3.3 represents two common elements in an aspect, advice and pointcuts, and is an example of a valid CAL aspect definition. This definition generates an abstract syntax tree via the use of a customized SAX2 [3] parser, in this case generating one public pointcut named DEPOSITMADE.

Figure 3.4 shows the AspectJ version of the aspect presented in Figure 3.3. While this thesis is predominately based on an analysis of AspectJ, and as such the semantics of the XML format essentially match the features of this language, CAL is flexible in the sense that it can generate

```
<cal>
    <aspect>
        <accessSpecifier>public</accessSpecifier>
        <name>AccountAspect</name>
        <pointcut>
           <accessSpecifier>public</accessSpecifier>
           <name>depositMade</name>
           <parameter>
              <type>void</type>
           </parameter>
           <pointcutExpression>
              <executionPointcut>
                <accessSpecifier>public</accessSpecifier>
                <returnType>star</returnType>
                <className>Account</className>
                <methodName>deposit</methodName>
                <argument>double</argument>
              </executionPointcut>
          </pointcutExpression>
        </pointcut>

        <advice>
          <adviceDeclaration>
             <afterAdvice>
                <parameter>
                   <type>void</type>
                </parameter>
             </afterAdvice>
          </adviceDeclaration>
          <pointcutSpecificationExpression>
             <pointcutSpecification>
                <name>AccountAspect</name>
                <argument>void</argument>
             </pointcutSpecification>
          </pointcutSpecificationExpression>
          <adviceBody>
             System.out.println(
               "A deposit has been made.");
          </adviceBody>
        </advice>
    </aspect>
</cal>
```

Figure 3.3: Example XML representation of an aspect written in AspectJ.

```
public aspect AccountAspect
{
   pointcut depositMade() :
      execution(* Account.deposit(double));

   after() : depositMade()
   {
      System.out.println("A deposit has been made.");
   }
}
```

Figure 3.4: Example aspect program written in AspectJ.

weaving rules that are dramatically different from AspectJ (e.g., few or many legal join points, unique weaving rules, unique precedence, unique keywords for weaving conditions). Further, while not provided in this work, aspects in other AOP languages can be translated, through typical source transformation techniques, into the XML format presented here, essentially making them usable by CAL.

3.1.2 Meta-Weaver Components

The meta-weaver portion of the design consists of five components: the *weaver*, the representation of the *weaving rules*, the *abstract syntax tree definitions* of the *base* and *aspect* programs, and the *woven abstract syntax tree*. The weaving rules and abstract syntax tree definitions are especially critical, as they provide CAL with modularity and flexibility.

On its own, the weaver is a mediator; it interprets any rules specified in the XML and loads algorithms representing these rules. This is accomplished through an *algorithm registry*, essentially a store of known AOP algorithms. When the weaver sees XML and notices a given tag (e.g., the name of a precedence algorithm), the weaver checks its algorithm registry to see if an implementation of that algorithm is available. Assuming CAL is given legal commands (otherwise it fails gracefully), it traverses the base and aspect abstract syntax trees to determine where the join point shadows will be placed. These locations are determined by the loaded rules and eventually output a combined base program and advice block. Figure 3.5 demonstrates the process of combining two

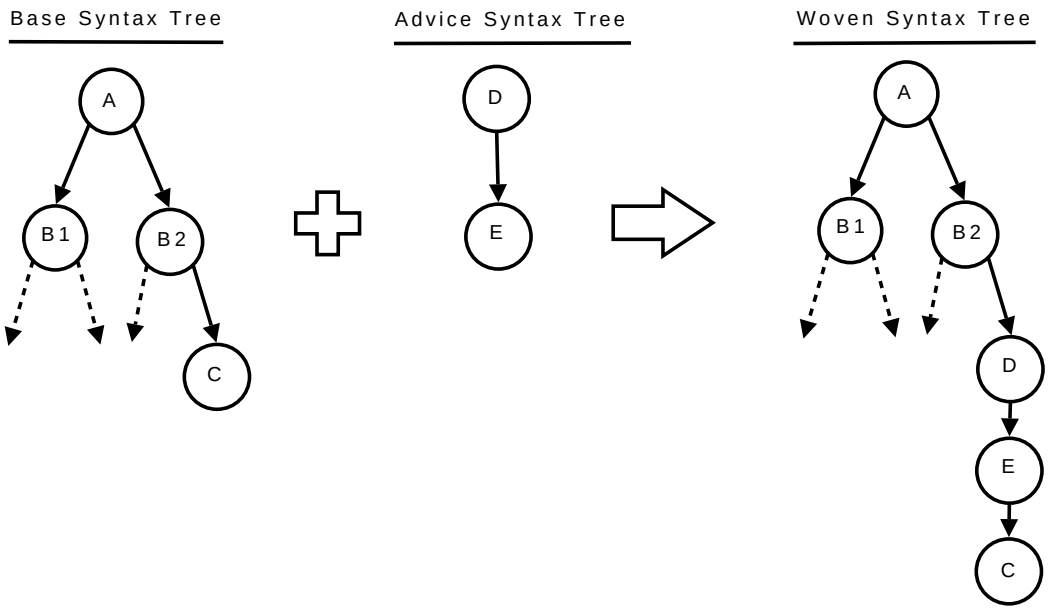

Figure 3.5: Weaving two abstract syntax trees using weaving rules produces a abstract syntax tree that combines the advice and base programs.

abstract syntax trees, assuming the join point shadow is located after the method B2.

The next component is the representation of weaving rules. Weaving rules represent all legal information about an AOP language, including what join points and pointcuts are available on the system, what the syntax is for the language, and the algorithms for weaving and precedence. Weaving rules interact with the algorithm registry, loading algorithms and join points into CAL on demand. Eventually these rules have an effect on the abstract syntax tree output by the system.

Once the abstract syntax tree is generated, by a third party program, the weaver can traverse the tree to determine the join point shadows. A visitor pattern is used to do the traversal of this tree. The weaver then identifies possible locations for weaving and inserts join point shadows into the base program's abstract syntax tree.

Besides the base code, CAL also represents abstract syntax trees for aspects. With the base code, third party programs, not discussed in this thesis, parse the code into a custom abstract syntax tree representation. With aspect abstract syntax trees, however, XML is parsed. This parser reads in elements (e.g., execution, call, before, around) of an aspect and builds an abstract syntax

```
public class Account {
    private double balance;
    public void deposit(double amount){
        balance += amount;
        System.out.println("A deposit has been made.");
    }
    public void printBalance(){
        System.out.println("Balance: " + balance);
    }
}
-----------------------------------------------
public class Main {
   public static void main(String[] args) {
      double amount = 100.05;
      Account account = new Account();
      account.deposit(amount);
      account.printBalance();
   }
}
```

Figure 3.6: Example base program written in Java.

tree that exactly mimics those elements. While this abstract syntax tree is being created, CAL caches pertinent information about the construction of the aspect. For example, CAL keeps a store of all pointcuts for easy retrieval.

Once the weaving rules, aspect, and base code are loaded, CAL can weave, producing the final component in question, the combined aspect + base abstract syntax tree. This tree can be translated back into source code via a custom visitor [10]. Each element is translated back into text which represents the original language of the base program, but now with advice inserted at the appropriate locations. Figure 3.6 shows an example of the woven output produced by CAL. This example is woven from the base code in Figure 3.2 and the aspect program in Figure 3.4.

```
aspect AccountAspect
{
   advice execution("% Account::deposit(...)"): after()
   {
      cout << "A deposit has been made." << endl;
   }
};
```

Figure 3.7: Example aspect program written in AspectC++.

3.2 CAL Aspects

In this section, I discuss aspects in CAL which are represented via an XML based wrapper of pointcuts, join points, and advice. While the previous section gave an overview of CAL, this section provides more detailed examples of the different components involved in weaving and is broken into three subsections: CAL aspects in XML, abstract syntax tree definition for aspects, and abstract syntax tree definition for base code and advice.

3.2.1 CAL Aspects in XML

By design, meta-weavers need to accommodate the functionality from a potentially large variety of aspect-oriented languages. Traditionally, these languages are written via custom parsing for a particular programming language [22]. Ironically, however, aspects are, in a sense, a cross-cutting concern in and of themselves. The rules used in languages like AspectJ are tied directly to Java's implementation (weaving into Java bytecode), despite the tremendous commonalities between language implementations (e.g., they have methods, variables, activation records, call stacks, type systems, conditionals, loops, etc.).

Generally speaking, a meta-weaver should abstract these rules, allowing them to be easily created and modified. To push us further toward this goal, CAL represents information about aspects as text, using XML, allowing aspects written in CAL to be easily modified despite the language, or set of aspect rules, in use. To accomplish this, an XML schema, shown in Appendix B.1, was created to allow different aspect languages to be translated into this XML format. Once the aspect

```
<cal>
    <aspect>
        <accessSpecifier>public</accessSpecifier>
        <name>AccountAspect</name>
        <pointcut>
           <accessSpecifier>public</accessSpecifier>
           <name>depositMade</name>
           <parameter>
              <type>void</type>
           </parameter>
           <pointcutExpression>
              <executionPointcut>
                <accessSpecifier>public</accessSpecifier>
                <returnType>star</returnType>
                <className>Account</className>
                <methodName>deposit</methodName>
                <argument>double</argument>
              </executionPointcut>
          </pointcutExpression>
        </pointcut>

        <advice>
          <adviceDeclaration>
             <afterAdvice>
                <parameter>
                   <type>void</type>
                </parameter>
             </afterAdvice>
          </adviceDeclaration>
          <pointcutSpecificationExpression>
             <pointcutSpecification>
                <name>AccountAspect</name>
                <argument>void</argument>
             </pointcutSpecification>
          </pointcutSpecificationExpression>
          <adviceBody>
             cout << "A deposit has been made."
                  << endl;
          </adviceBody>
        </advice>
    </aspect>
</cal>
```

Figure 3.8: Example XML representation of an aspect written in AspectC++.

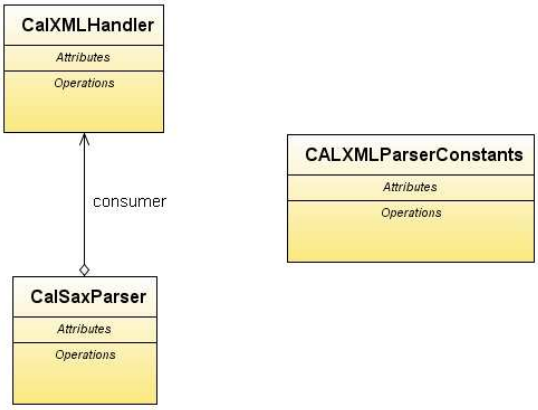

Figure 3.9: UML diagram of the custom XML parser for the aspect representation.

program is in the XML format, it can be parsed by a SAX2 parser, which builds the abstract syntax tree and outputs woven code.

Consider the aspect shown in Figure 3.4, which is written in AspectJ. This aspect identifies the join point after the execution of the method DEPOSIT in the class ACCOUNT. In contrast, consider the aspect in Figure 3.7, written in AspectC++. Both aspects identify the join point location after the execution of the method DEPOSIT in the class ACCOUNT, but do so via different syntax. These two aspects are functionally the same, though they each reflect the languages they are associated with (e.g., Java and C++). Translating both of these aspects into the XML format results in similar XML representations, shown in Figure 3.3 (Java) and Figure 3.8 (C++). The only difference between the two translations is the advice body, which contains code that is written in either Java or C++.

3.2.2 Abstract Syntax Tree Definition for Aspects

Once CAL receives an XML based representation of an aspect, it parses and transforms the text into an abstract syntax tree. This tree encapsulates the advice, pointcuts, and other attributes, informing the meta-weaver using whatever custom rules it currently has loaded, when, where, and how advice should be woven into the base code.

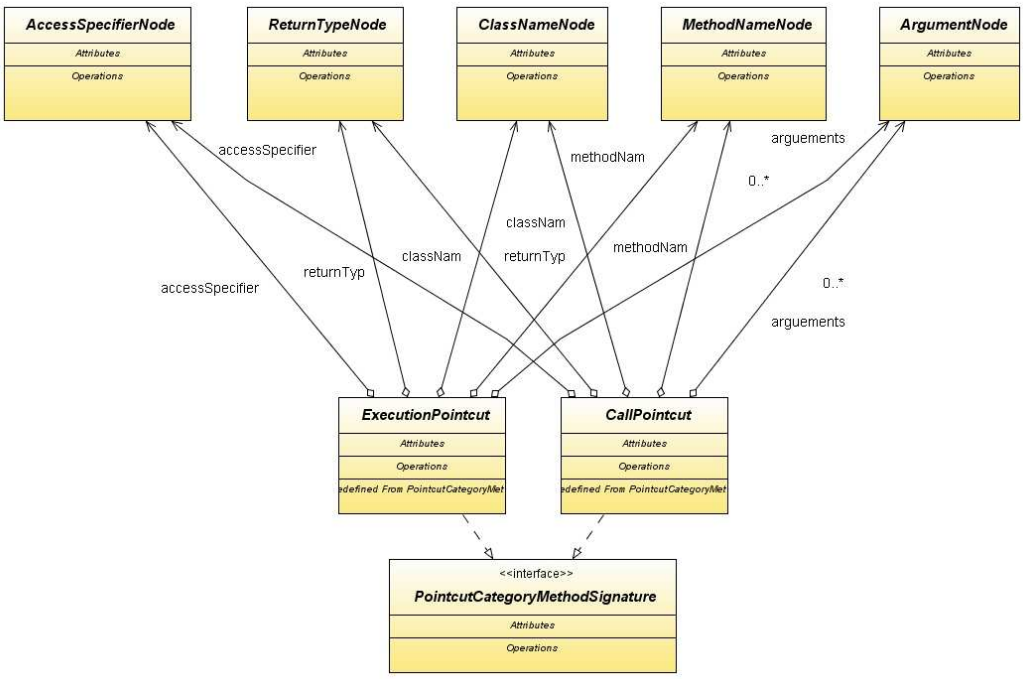

Figure 3.10: UML diagram of the aspect representation of pointcuts.

The first step in building the abstract syntax tree for aspects was to build a custom XML parser based on SAX2. Figure 3.9 shows the UML for the tree read in by this parser. This tree is created by identifying each start element as it is read in, then generating that element using a NODEFACTORY, shown in Figure A.5. This process uses a CALSAXPARSER to parse the file and a CALXMLHANDLER to process the parsed elements. CALXMLHANDLER generates each element, sets the attributes of each element, and determines the appropriate relationship between the elements. In other words, CALXMLHANDLER is responsible for generating the abstract syntax tree.

The NODEFACTORY is responsible for generating each element, or more appropriately, each node for our abstract syntax tree. This is accomplished by instantiating each class that corresponds to the element which was parsed by a CALSAXPARSER. In Figure A.5, these classes all inherit from the class NODE and represent individual components of an aspect.

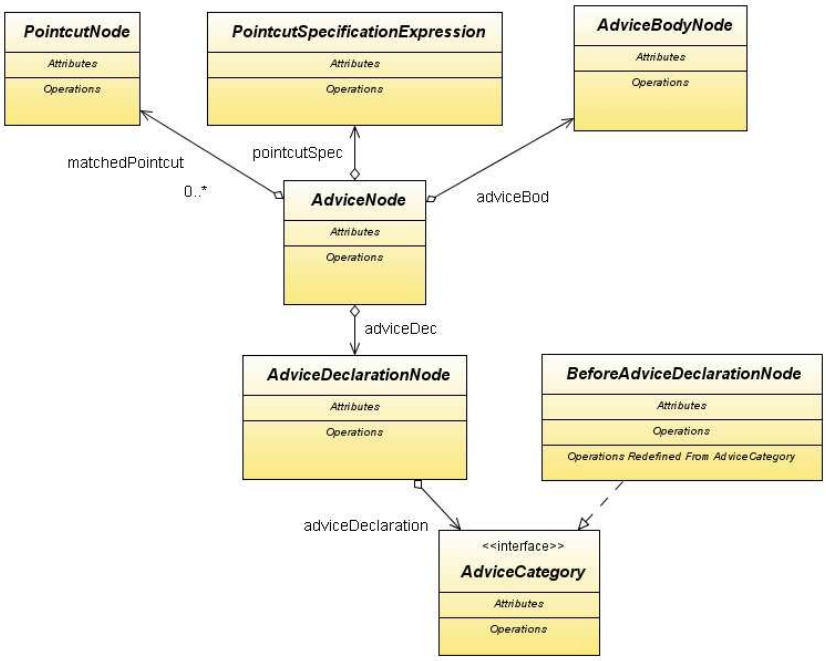

Figure 3.11: UML diagram of the aspect representation of advice.

In Figure 3.10, we see some of the classes of the aspect abstract syntax tree. The class structure in this figure represents pointcuts. The interface, POINTCUTCATEGORYMETHODSIGNATURE, sets up the functionality of all pointcuts which use a method signature to identify the join point. Similar interfaces are set up for other types of pointcuts which use a constructor signature, type signature, or field signature. In addition, the pointcuts are broken down into smaller parts, RETURNTYPENODE, CLASSNAMENODE, METHODNAMENODE, ARGUMENTNODE, and ACCESSSPECIFIERNODE, representing the parts of the CALLPOINTCUT and the EXECUTIONPOINTCUT. In Figure 3.11, advice is similarly broken down into its constituent components: advice type, pointcut specification, and advice body.

3.2.3 Abstract Syntax Tree Definition for Base Code and Advice

Both the base code and advice body use the same intermediate representation of source code. This intermediate is shown in Figure 3.12. While the predominant part of this thesis is on the design of

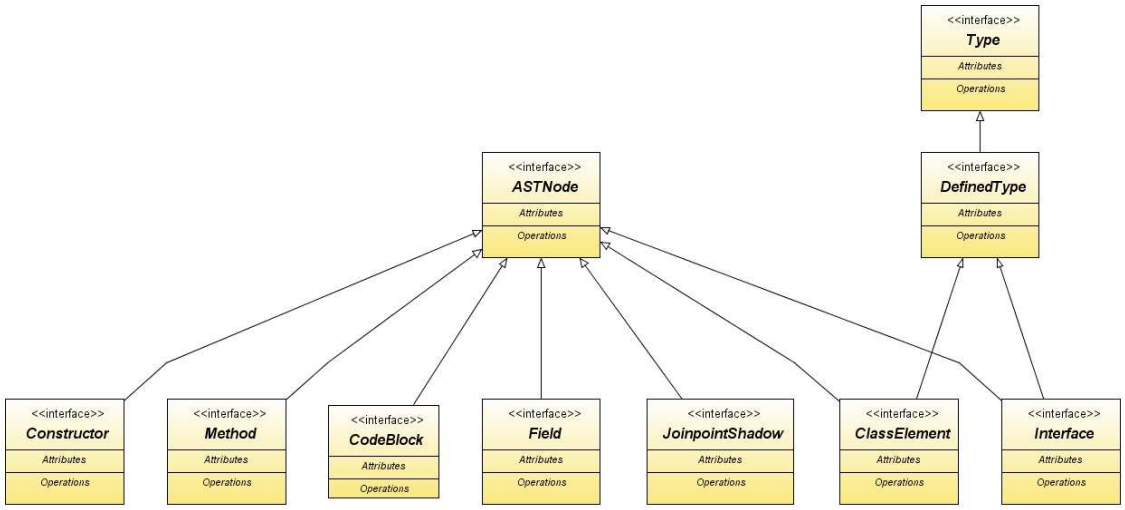

Figure 3.12: UML diagram of the base program representation.

the meta-weaver and the aspect XML representation, this section briefly discusses the base code representation used by CAL.

The classes in Figure 3.12 all inherit from ASTNODE and represent a node that can be identified by a pointcut. For example, a call or execution pointcut identifies a METHOD as the location where the advice will be woven. In AspectJ, we can identify FIELD, METHOD, TYPE, and CONSTRUCTOR as nodes in the base code's abstract syntax tree which can be identified as join points. This join point is identified in the tree by JOINPOINTSHADOW, essentially a marker for a join point.

While Figure 3.12 does not represent the entire abstract syntax tree, it does represent the nodes in the tree which are required to identify certain join points. To accommodate different implementations of the classes in the AST, in part because a different researcher is creating part of the implementation, CAL includes an abstract factory [10] that allows different AST implementations to be swapped dynamically.

CHAPTER 4

META-WEAVER DESIGN

The meta-weaver is designed to handle different approaches to weaving aspects into a base program (e.g., join point shadows, morphing aspects [17]). For this process to be flexible, the weaver must provide an interface that allows certain attributes to be changed. This is precisely what the weaver component of the meta-weaver accomplishes using the weaving rules to determine how to configure itself. To make this process understandable, the format and functionality of the weaving rules will be discussed. Then I will explore the process of combining these rules to create the custom weaver component.

4.1 Weaving Rules

Figure 4.1 shows an example set of XML weaving rules that are used to construct a custom weaver. There are three types of rules which are used to define this custom weaver: *join points*, the *weaving algorithm*, and *precedence rules*. Once these rules have been discussed, I will discuss the *algorithm registry*, a method CAL uses to allow new rules to be created.

4.1.1 Join Point Rules

The first type of weaving rule is JOINPOINT. Join point rules allow a developer to identify which join points can be used when performing the weave and are specified by the elements NAME, ASTNODE, and WHEN. The NAME element specifies what the syntax is for calling the pointcut. For example, in AspectJ, a pointcut's name could be EXECUTION, CALL, etc. The ASTNODE element identifies the node in the base abstract syntax tree where the join point will be attached (e.g., METHOD, CONSTRUCTOR). Lastly, the WHEN element identifies, relative to the ASTNODE, where to place the join point; in AspectJ, and also currently in CAL, the options are *before*, *after*, or *around*.

These three elements allow a developer to control join point shadow locations and syntax used

```
<calrules>
    <joinpoint>
        <name>execution</name>
        <ASTNode>Method</ASTNode>
        <when>before</when>
    </joinpoint>
    <weavingalgorithm>
        AspectJ
    <weavingalgorithm>
    <precedence>
        OrderInFile
    </precedence>
</calrules>
```

Figure 4.1: XML representation of weaving rules in CAL.

by the aspect-oriented language with only four XML tags. By creating an XML language where any node in the base code can be included as part of the weaving rules, the designer can implement new functions of an AOP language. For example, suppose a developer wanted to define a custom join point to execute before every code block (e.g., before an if statement, while loop on each iteration, method). Doing so in CAL is trivial. The only required change would be to the tag in Figure 4.1 from $< ASTNode >$METHOD$< /ASTNode >$ to $< ASTNode >$CODEBLOCK$< /ASTNode >$. This extremely minor change causes CAL to weave code in a dramatically different manner.

As a final example of join point rules, consider Figure 4.1, which allows advice to be woven *before* the *execution* of a *method*. Now, consider the aspect in Figure 4.2; the weaver would not weave the advice from this aspect, given the rules in Figure 4.1. This is because the aspect defines the join point as executing *after* a METHOD is finished executing, but the *after* join point is not defined under this set of weaving rules!

4.1.2 Weaving and Precedence Rules

CAL also has rules for custom weaving algorithms which can be swapped both statically and dynamically. CAL currently uses an algorithm named AspectJ, which uses join point shadows to

```
public aspect Example
{
   after() : execution(* Account.deposit(..))
   {
      System.out.println("A deposit has been made.");
   }
}
```

Figure 4.2: An aspect which specifies a join point after the execution of the method DEPOSIT.

identify possible weaving locations. This algorithm is similar to the weaving algorithm used in AspectJ 1.1 and is discussed in more detail in Hilsdale and Hugunin [19].

In addition to the weaving algorithm, precedence rules can be specified in the XML document for weaving rules. Precedence identifies a specific order in which each piece of advice is woven. Similar to the weaving algorithm element, the precedence is given a name which is used by the weaver to identify which precedence rule to use. In Figure 4.1, the precedence rule that is used is ORDERINFILE, which will weave the join points in the order of the advice in the aspect. Suppose, for example, an aspect defines three pointcuts, A, B, and C, all of which are asked to weave at a given location. Using the ORDERINFILE tag, CAL would give the highest precedence to the pointcut that is closest to the top of the file (e.g., the first pointcut read in by CAL). Other possible precedence tags are Alphabetical, AspectJ, or whatever other custom precedence rule is desired.

4.2 Weaver

The weaver in CAL has two primary components: the JOINPOINTREGISTRY and the ALGORITHMREGISTRY. In this section, I discuss how the weaver incorporates these registries and uses them to actually weave advice into base code.

4.2.1 Join Point Registry

To illustrate this process, consider that CAL has read in a series of weaving rules similar to that shown in Figure 4.3. CAL needs to somehow translate this XML into an actual set of algorithms and procedures for implementing an AOP language.

How does CAL accomplish this? First, when CAL sees a JOINPOINT element in a weaving rules file, it registers a particular join point rule under an ASTNODE's name into the JOINPOINTREGISTRY, a component for tracking information on the valid join points in the system. The weaving algorithms will then consult this registry to determine what join points are available, and where advice can be placed relative to those join points. While Figure 4.3 is an example of an AOP language with only one join point, this is highly atypical and, as such, CAL allows many join points to be loaded and registered with the environment. This is accomplished by creating a subclass of JOINPOINT and adding it to the JOINPOINTREGISTRY by calling .ADD(JOINPOINT JOINPOINT).

4.2.2 Algorithm Registry

The algorithm registry is similar to the join point registry, but the algorithm registry tracks the current weaving algorithm and precedence rule. Unlike the join points there can only be one weaving algorithm and precedence rule. When CAL begins the weaving process, it references what is called the ALGORITHMREGISTRY. This class represents a list of known algorithms that can be used by CAL. As an example, there are two steps for creating a new weaving algorithm. First, one needs to implement a subclass of the WEAVINGALGORITHM class. Second, you need to add the algorithm to the ALGORITHMREGISTRY by calling .LOAD(STRING TYPE, OBJECT O). The same can be done for precedence rules.

Once the algorithm registry has been notified of the new algorithm, the CAL meta-weaver rules can use the algorithm as a tag in an XML file. When CAL needs to use the current precedence rule, it queries the algorithm registry with a .LOOKUP(STRING NAME) call, passing in the name of the algorithm it is looking for (in this case, the tag PRECEDENCE). In short, this querying process allows a designer using CAL to not only create new precedence rules, but to make new AOP algorithms, allowing future developers to use CAL's architecture to customize the weaving process.

```
<calrules>
    <joinpoint>
        <name>ifStatement</name>
        <ASTNode>IfStatement</ASTNode>
        <when>after</when>
    </joinpoint>
    <weavingalgorithm>
        AspectJ
    <weavingalgorithm>
    <precedence>
        OrderInFile
    </precedence>
</calrules>
```

Figure 4.3: A CAL weaving rule that defines a new join point for an IF statement.

4.3 Putting it all Together

In this section, I explore how CAL takes an XML specification and translates it into a functioning weaver, providing insight into how CAL connects all of these separate architectures. I discuss this *putting it all together* section from two angles. First, I present Java code that is similar to the process CAL uses when reading in custom rules. Second, I provide an example demonstrating how CAL creates an aspect-oriented programming language that functions like AspectJ.

4.3.1 Connecting the CAL Components

This section discusses how CAL uses its registries to manipulate and control both weaving rules and algorithms, connecting CAL's components. The basic idea is that CAL delegates weaving operations to algorithms that are pre-loaded into the CAL registries. When a user then gives a weaving rules file, the user is specifying which of these pre-loaded algorithms they would like to use (e.g., a certain precedence algorithm, weaving algorithm, or set of join points).

As an example, consider Figure 4.1. When CAL loads this XML file, its registries are checked for each of the rules in that file and these are dynamically loaded into the current weaver's individual registry. In essence, CAL generates code similar to that shown in Figure 4.4. This code updates the registries and specifies which rules the weaver will use.

This process begins by setting up the registries, ALGORITHMREGISTRY and JOINPOINTREGISTRY. Once the registries have been loaded with the appropriate algorithms, CAL parses the join point rules and generates the corresponding rule object. In this case the JOINPOINT object named METHOD is initialized to the values read in from the XML file. Then the weaving algorithm and precedence rule objects, MYWEAVINGALGORITHM and MYPRECEDENCECHOICE, are generated based on the weaving rule file.

Once the weaving rule file has been loaded into CAL, the ALGORITHMREGISTRY (named REGISTRY in Figure 4.4) loads both the weaving algorithm and precedence rule. Then the JOINPOINTREGISTRY (JPREG in Figure 4.4) loads any of the join points that were created. In this case, there is only one join point, an EXECUTION join point that happens BEFORE a METHOD is called.

However, code that interacts with the arbitrarily chosen join points in a user's XML requires that pointcut subclasses be created that define the semantics of the pointcut (e.g., a subclass for execution, call, etc). For example, if a developer created a new join point rule with the elements DEEPRECURSION (weaving occurs when methods have been recursively called many times), Method, and After, they would need to implement the pointcut. This can be accomplished by creating a subclass of POINTCUTCATEGORYMETHODSIGNATURE and calling it DEEPRECURSION.

4.3.2 Building AspectJ in CAL

Consider a developer who wants to create, in CAL, a close replica of the programming language AspectJ. The standard algorithm for weaving is that given by Hillsdale [19], which for the sake of discussion I give the name "AspectJ." Similarly, the precedence algorithm could be named ASPECTJPRECEDENCE.

If implementations of the AspectJ precedence and weaving algorithms are not available, the programmer must then write them into CAL by creating a subclass of the PRECEDENCERULE and ALGORITHMRULE classes. These rules must be registered in the ALGORITHMREGISTRY. Once this implementation is complete, the user then has to write, into the XML file, join points for every

```
//setup the meta-weaver
MetaWeaver weaver = new DefaultMetaWeaver();

//setup the registries
AlgorithmRegistry registry = weaver.getAlgorithmRegistry();
JoinPointRegistry jpReg = weaver.getJoinPointRegistry();

//make one join point
JoinPoint method = new JoinPoint();
method.setName("Execution");
method.setASTNode("Method");
method.setBefore(true);

//determine weaving algorithm
AspectJ1a1 myWeavingAlgorithm = new AspectJ1a1();

//determine precedence rule
OrderInFile myPrecedenceChoice = new OrderInFile();

//load rules into algorithm registry
registry.load(AlgorithmRegistry.PRECEDENCE, myPrecedenceChoice);
registry.load(AlgorithmRegistry.WEAVING, myWeavingAlgorithm);

//load legal join points into the join point registry
jpReg.add(method);

//weave
ASTNode wovenCode = weaver.weave(basecode, aspects);
```

Figure 4.4: How CAL connects up the weaving rules and join points.

program location used by AspectJ, including those for calling or executing methods as well as others. These join points, similarly, must be registered with the JOINPOINTREGISTRY.

So what happens if a user tries to use AspectJ algorithms that are not registered? If a user tries to use an algorithm not registered by CAL (e.g., they type a random word into the precedence field), CAL will attempt to lookup the faulty rule, will fail, and will inform the user of an error. For example, if a language designer wanted to experiment with AspectJ by disabling execution pointcuts, they could accomplish this rather easily — by removing execution join points from the JOINPOINTREGISTRY, elevating this join point's use in a pointcut to a syntax error. This essentially means that, before any advice is woven, weavers must inspect the join point registry for a given weaving rule, ensuring that the users request for *before*, *after*, or *around*, is allowed on that join point, and also ensuring that weaving on that node is allowed at all.

While CAL's procedure for analyzing and weaving source code may appear, on the surface, to be rather complex, the end result is a customizable platform for experimenting with aspect-oriented programming languages. CAL uses a system of *registries* for join points, weaving algorithms, and precedence rules, allowing the programmer to influence virtually any component in an intermediate representation of source code.

CHAPTER 5

CONCLUSION

To promote experimentation, I have developed a meta-weaver framework that enables a developer's ability to customize aspect-oriented languages and weaving rules for those languages. The CAL meta-weaver is a step toward this because it provides both a framework for modifying the features of an aspect-oriented language and a framework to modify the way transformations of aspects occur. CAL accomplishes this through the use of XML, which gives the developer a great deal of control over the structure of the generated code. Through this control, we can empirically compare combinations of aspect-oriented language features and weaving rules.

5.1 Significance and Claims

In this thesis, a meta-weaver framework has been implemented and described. This framework allows the weaving of an aspect-oriented program with an existing base program, given user defined rules. This work further expands on the work of Rohlik, et.al. [33] and Gray [15], whose, key goal was also to enable experimentation with aspect-oriented languages via the creation of a highly customizable weaving architecture.

Rohlik, et.al. built the Xweaver [33, 16], which is an open source framework that uses rules defined in an XSL program to customize the process of weaving. These rules allow new types of transformations, from an individual aspect to the base program, to be defined. In contrast, CAL not only allows us to customize these transformations but also allows the weaving rules to control precedence, or ordering of the aspect transformations. Another key difference is that the Xweaver's target language is limited to C++. However, CAL is flexible enough to accommodate any language which is represented in the abstract syntax tree for the base program.

CAL also uses a framework which is similar to the meta-weaver in Gray's work [15], though Gray's work does not have a framework for weaving rules. Gray's meta-weaver is designed to

generate weavers, at a high level of abstraction, each accommodating a different language. Ideally, each new weaver that is generated minimizes the duplication of components. This is because each newly generated weaver presumably shares some of the features of the previously generated weavers. The CAL framework, on the other hand, does not generate a program, but provides a framework where the base program's abstract syntax tree can be easily built up and modified to accommodate different languages.

The CAL meta-weaver framework provides an environment in which AOP properties and weaving is easily customizable. With a broad number of aspect languages, each having its own model for aspect orientation, CAL fills a need for a framework that is flexible enough to allow each aspect feature to be quickly changed. Through this framework, we can enable experimentation of weaving algorithms, syntax, and semantics of aspect-oriented languages.

5.2 Future Work

We know relatively little about the effect of the AOP paradigm on programming. As such, CALs meta-weaver framework can be used to gather empirical evidence on how well the different AOP techniques work together. The scope of this testing includes human studies on comprehension of the effects of aspect orientation and the individual impact of syntax, semantics, or technical concerns related to algorithm efficiency. CAL can be used to support all of these types of studies.

In addition, meta-weavers allow us to more easily evaluate the interactions between weaving rules and the resulting software. For example, Schonger et.al. [34] suggest that pointcuts in AspectJ fall into classes which are not precisely defined and can be syntactically unclear. However, does this AOP minutia have any real, significant effect on human performance when using an AOP language? Using CAL allows us to easily test this hypothesis in experimental studies involving humans.

APPENDICES

APPENDIX A

UML DIAGRAMS OF CAL

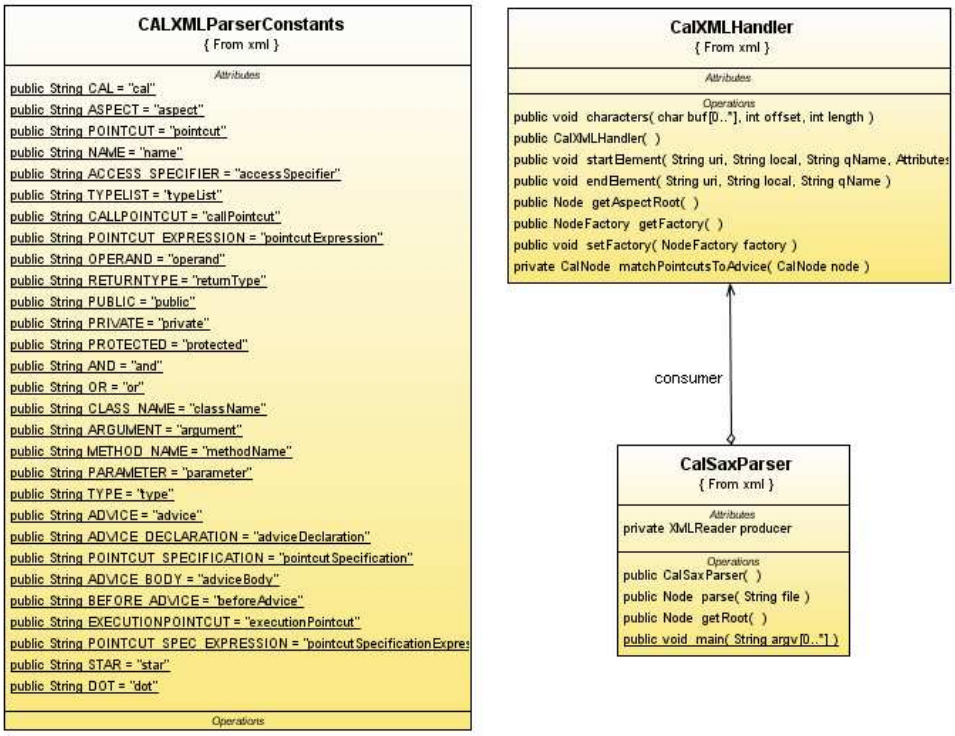

Figure A.1: Diagram of the SAX parser design.

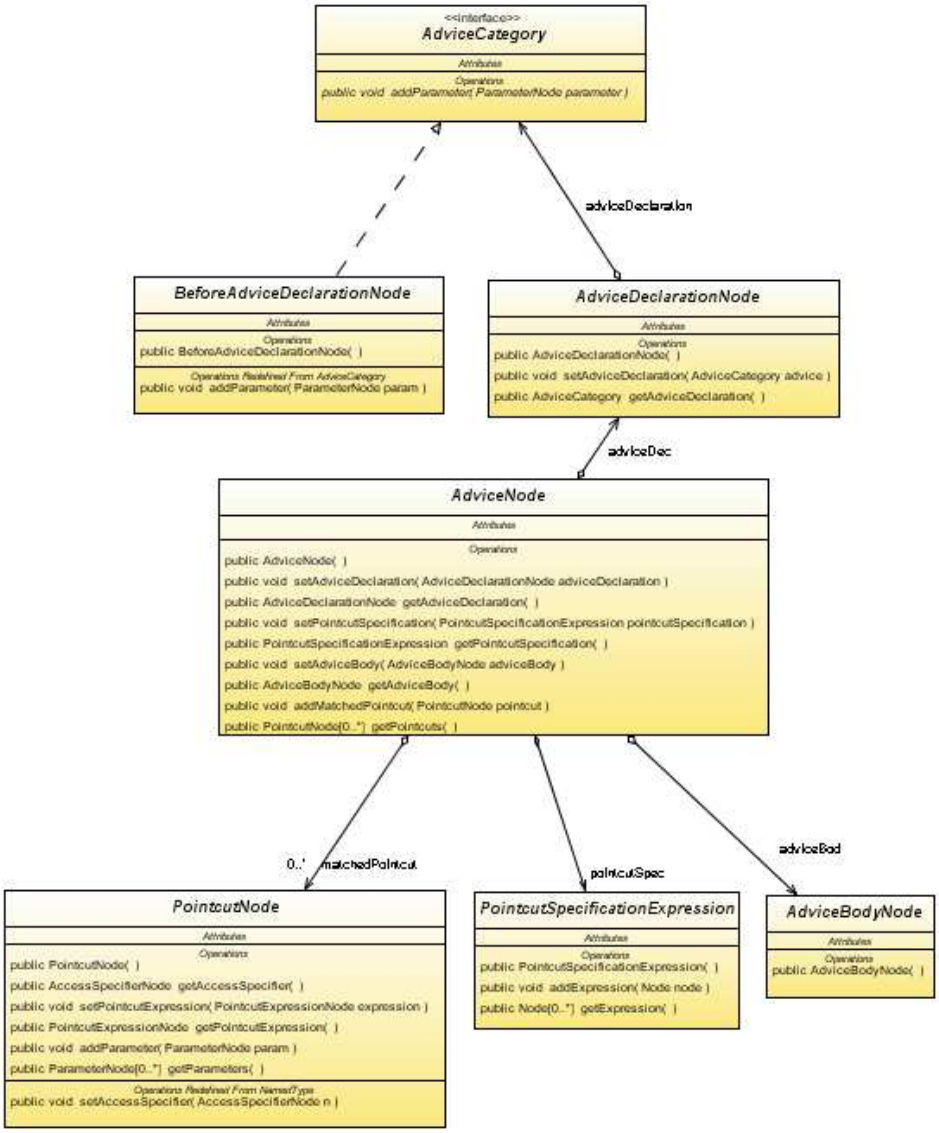

Figure A.2: UML diagram of advice in the aspect syntax tree definition.

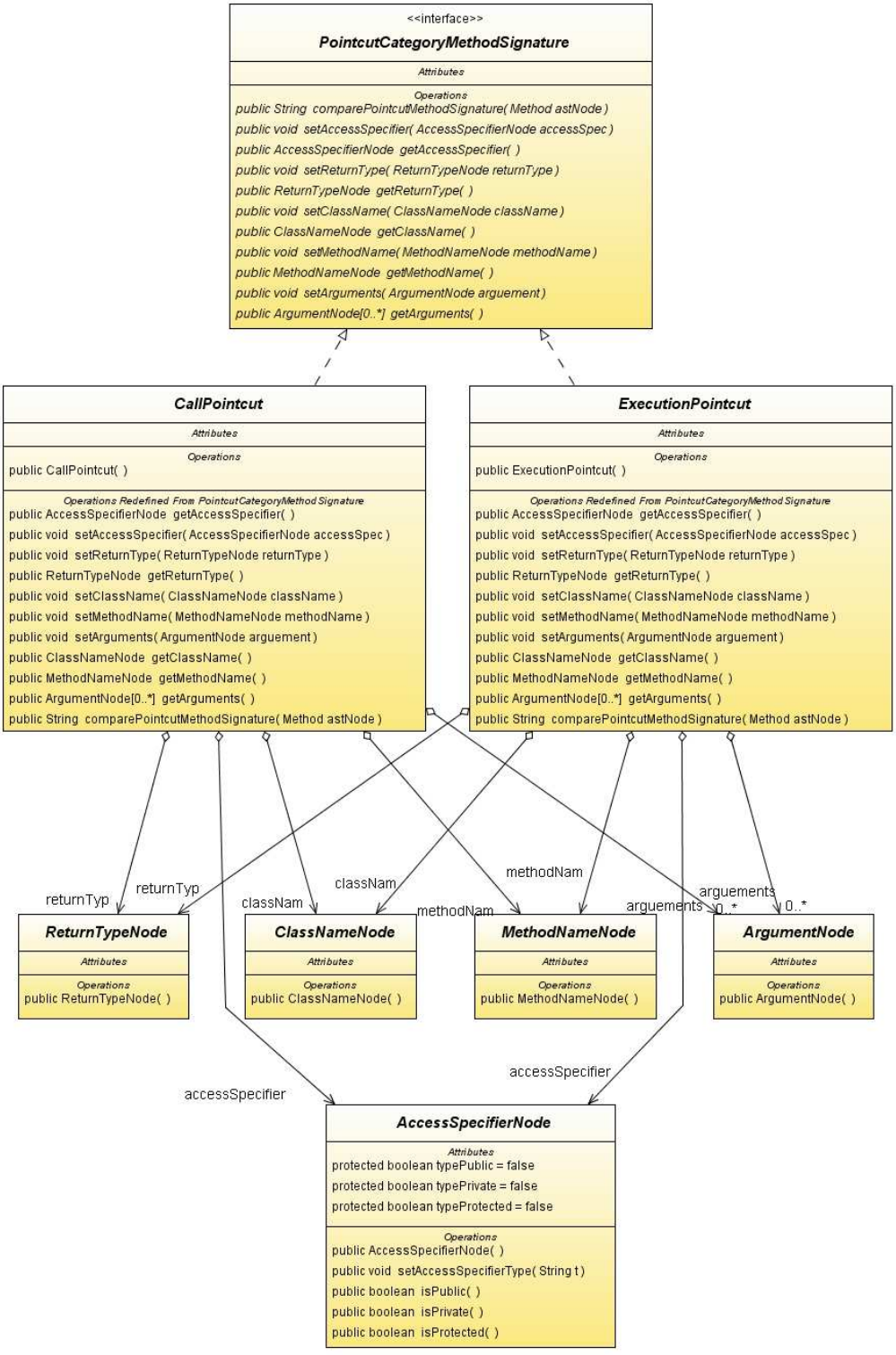

Figure A.3: UML diagram of pointcuts in the aspect syntax tree definition.

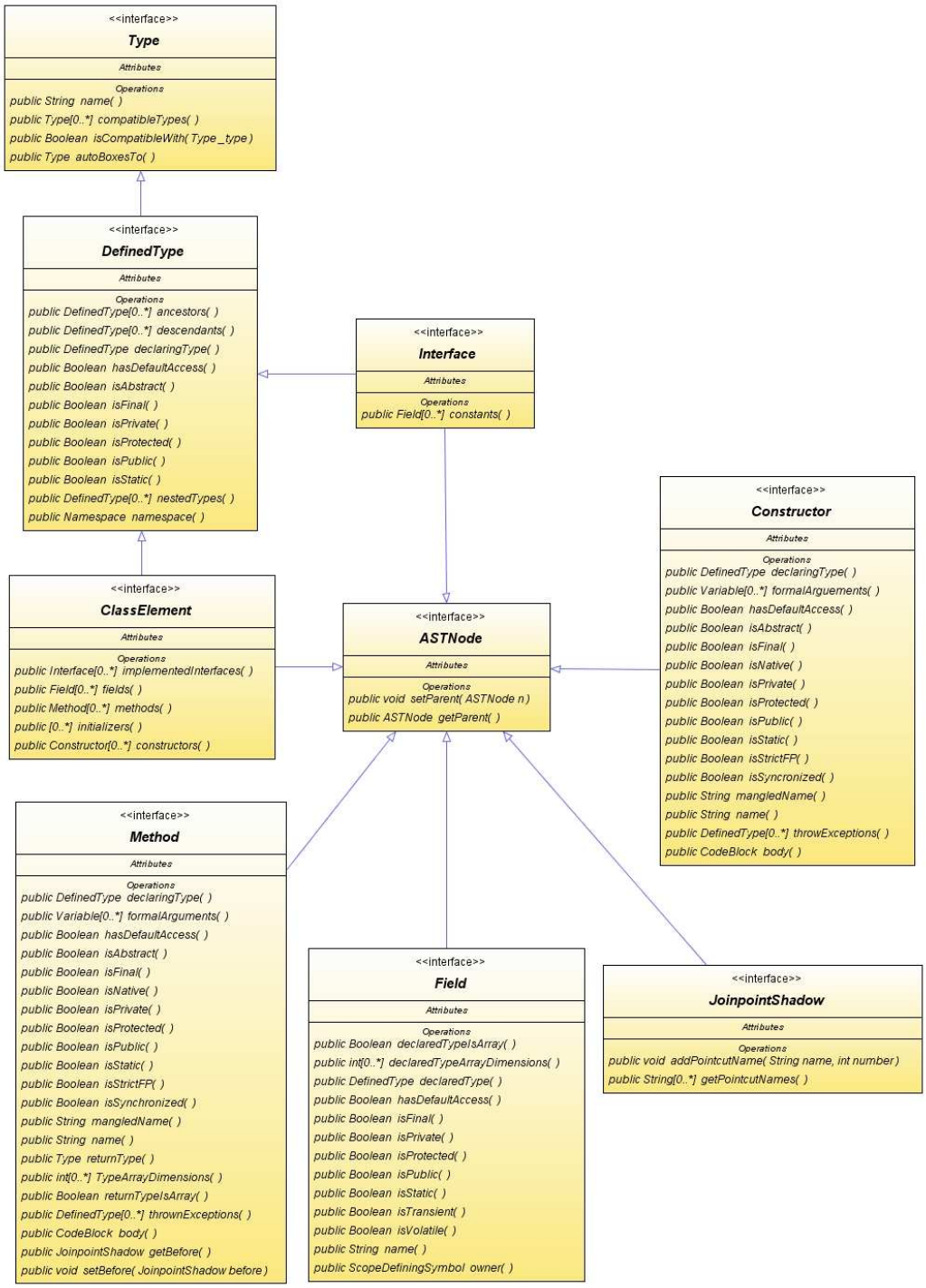

Figure A.4: UML diagram of base program representation.

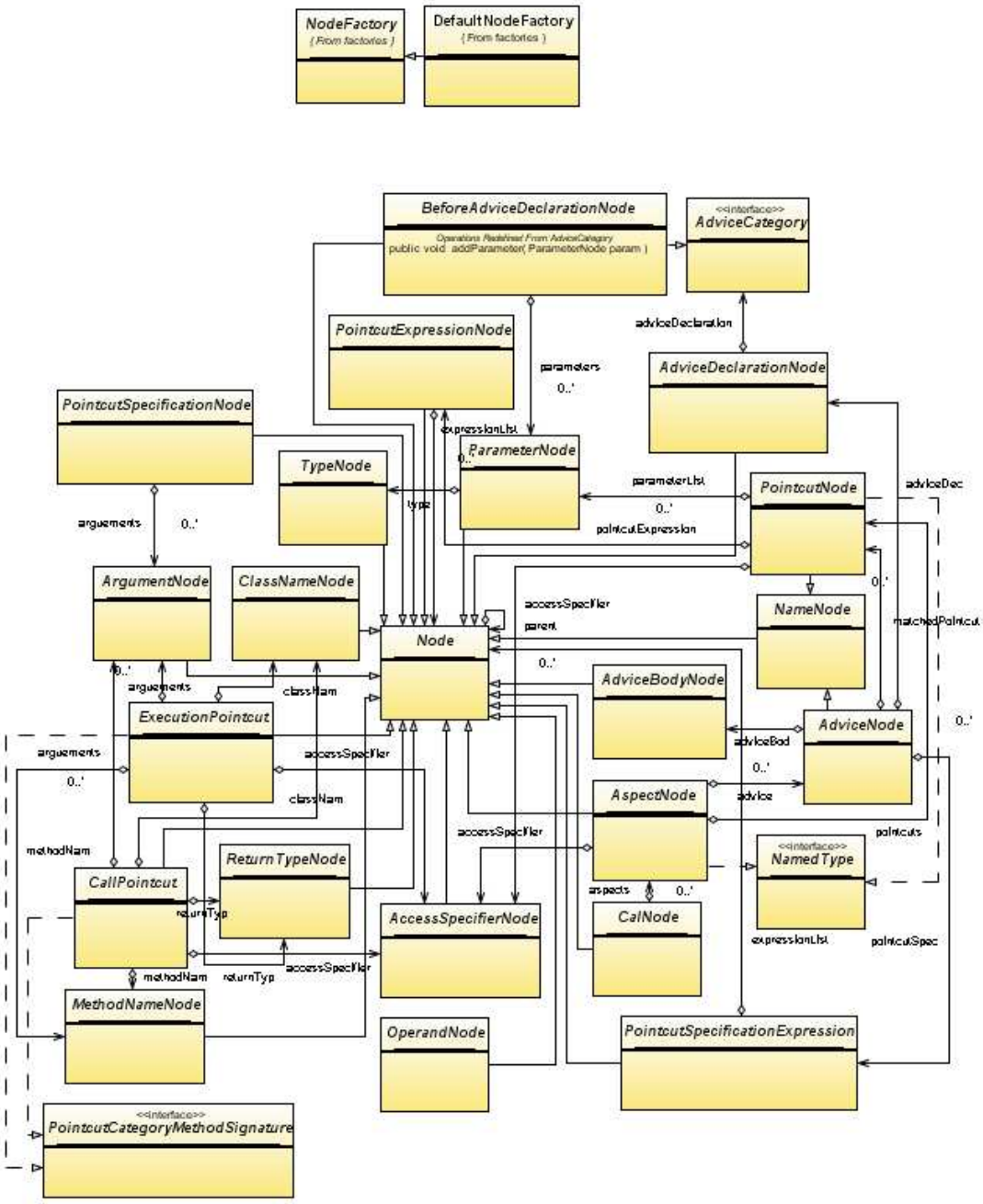

Figure A.5: UML diagram of the syntax tree definition for aspects.

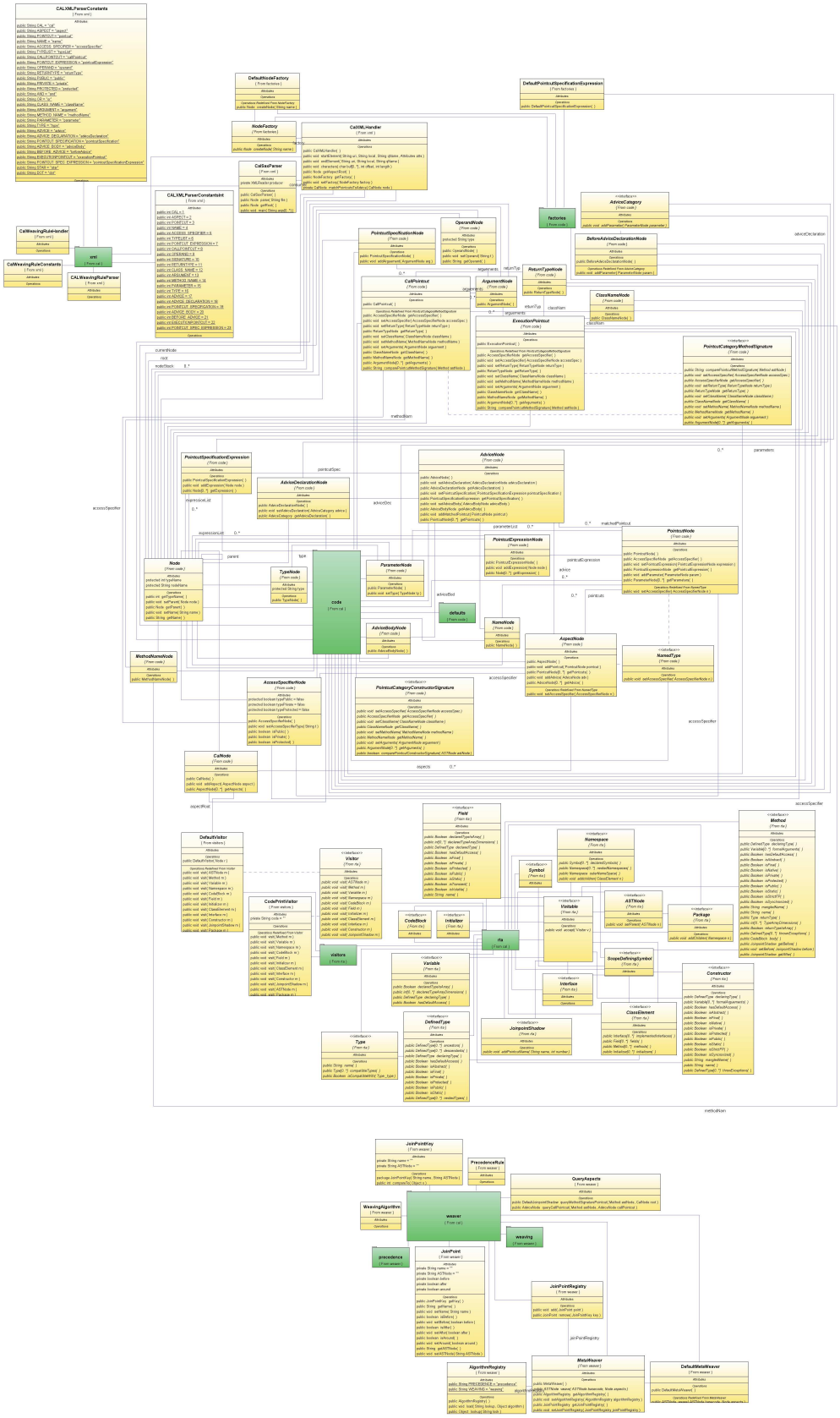

Figure A.6: UML diagram of the CAL meta-weaver.

APPENDIX B

XML SCHEMA

B.1 XML Schema for Aspects

This first XML schema specifies the format of the XML file for aspects. This schema does not represent all of the possible aspect-oriented languages, but can be extended to represent any other aspect-oriented language.

```
<?xml version="1.0" encoding="UTF–8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="accessSpecifier">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="advice">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="adviceDeclaration" />
        <xs:element ref="pointcutSpecificationExpression" />
        <xs:element ref="adviceBody" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="adviceBody">
```

```
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="adviceDeclaration">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="beforeAdvice" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="argument">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="aspect">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="accessSpecifier" />
      <xs:element ref="name" />
      <xs:element ref="pointcut" maxOccurs="unbounded" />
      <xs:element ref="advice" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="beforeAdvice">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="parameter" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="cal">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="aspect" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="callPointcut">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="accessSpecifier" />
      <xs:element ref="returnType" />
      <xs:element ref="className" />
      <xs:element ref="methodName" />
      <xs:element ref="argument" />
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="className">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="executionPointcut">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="accessSpecifier" />
      <xs:element ref="returnType" />
      <xs:element ref="className" />
      <xs:element ref="methodName" />
      <xs:element ref="argument" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="methodName">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="name">
```

```
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="operand">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="parameter">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="type" />
      <xs:element ref="name" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="pointcut">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="accessSpecifier" />
      <xs:element ref="name" />
      <xs:element ref="parameter" />
      <xs:element ref="pointcutExpression" />
    </xs:sequence>
  </xs:complexType>
```

```
</xs:element>

<xs:element name="pointcutExpression">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="executionPointcut" />
      <xs:element ref="callPointcut" minOccurs="0" />
      <xs:element ref="operand" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="pointcutSpecification">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" />
      <xs:element ref="argument" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="pointcutSpecificationExpression">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="pointcutSpecification" maxOccurs="unbounded" />
```

```
        <xs:element ref="operand" minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="returnType">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="type">
    <xs:complexType mixed="true" />
  </xs:element>

</xs:schema>
```

B.2 XML Schema for Weaving Rules

This XML schema specifies the format of the XML file for weaving rules. This schema is able to represent any precedence rule or weaving algorithm. In addition, this schema represents any number of join point rules that correspond to the XML schema for aspects.

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ASTNode">
    <xs:complexType mixed="true" />
  </xs:element>
```

```
<xs:element name="calrules">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="joinpoint" maxOccurs="unbounded" />
      <xs:element ref="weavingalgorithm" />
      <xs:element ref="precedence" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="joinpoint">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" />
      <xs:element ref="ASTNode" />
      <xs:element ref="when" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="name">
  <xs:complexType mixed="true" />
</xs:element>
```

```
  <xs:element name="precedence">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="weavingalgorithm">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="when">
    <xs:complexType mixed="true" />
  </xs:element>

</xs:schema>
```

BIBLIOGRAPHY

[1] Jonathan Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. Technical Report CMU-ISRI-04-108, Carnegie Mellon University, 2004.

[2] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In *SPLAT '04: In workshop on Software Engineering Properties of Languages for Aspect Technologies*, Lancaster, UK, 2004.

[3] David Brownell. *SAX2*. O'Reilly and Associates, Inc., Sebastopol, CA, 2002.

[4] Nelio Cacho, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thais Batista, and Carlos Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 109–121, New York, NY, USA, 2006. ACM Press.

[5] Siobhán Clarke and Robert J. Walker. Composition patterns: an approach to designing reusable aspects. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Curtis Clifton and Gary Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-15, Iowa State University, 2003.

[7] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Reports, pages 33–44. Department of Computer Science, Iowa State University, April 2002.

[8] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, New Jersey, 1976.

[9] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

[11] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[12] Jeff Gray. Using software component generators to construct a meta-weaver framework. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 789–790, Washington, DC, USA, 2001. IEEE Computer Society.

[13] Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. An approach for supporting aspect-oriented domain modeling. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 151–168, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[14] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Commun. ACM*, 44(10):87–93, 2001.

[15] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[16] Iris Groher and Markus Voelter. Xweave: models and aspects in concert. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 35–40, New York, NY, USA, 2007. ACM.

[17] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 46–55, New York, NY, USA, 2004. ACM.

[18] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[19] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[20] Mik Kersten and Gail C. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 340–352, New York, NY, USA, 1999. ACM Press.

[21] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, 1996.

[22] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[23] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[24] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

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

[26] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[27] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 191–204, New York, NY, USA, 2006. ACM Press.

[28] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[29] Odysseas Papapetrou and George A. Papadopoulos. Aspect oriented programming for a component-based real life application: a case study. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1554–1558, New York, NY, USA, 2004. ACM Press.

[30] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[31] Meghan Revelle, Tiffany Broadbent, and David Coppit. Understanding concerns in software: Insights gained from two case studies. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3, 2007.

[33] O. Rohlik, A. Pasetti, P. Chevalley, and I. Birrer. An Aspect Weaver for Qualifiable Applications. In *Data System in Aerospace (DASIA) Conference*, Nice, France, July 2004.

[34] Stefan Schonger, Elke Pulvermüller, and Stefan Sarstedt. Aspect-oriented programming and component weaving: Using XML representations of abstract syntax trees, February 2002. Second German AOSD Workshop, Bonn, Germany. To appear.

[35] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM Press.

[36] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.

[37] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and

value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2001. ACM Press.