ISR Institute for Software Research
University of California, Irvine

The Aspect Oriented Markup Language and its Support of Aspect Plugins

Cristina Videira Lopes
University of California, Irvine
lopes@ics.uci.edu

Trung Chi Ngo
University of California, Irvine
trungcn@uci.edu

October 2004

ISR Technical Report # UCI-ISR-04-8

Institute for Software Research
ICS2 210
University of California, Irvine
Irvine, CA 92697-3425
www.isr.uci.edu

http://www.isr.uci.edu/tech-reports.html

The Aspect Markup Language and its Support of Aspect Plugins

Cristina Videira Lopes and Trung Chi Ngo
Institute for Software Research and Bren School of Information and Computer Sciences
Department of Informatics
University of California, Irvine
{lopes, trungcn} @ ics.uci.edu

ISR Technical Report # UCI-ISR-04-8

October 2004

Abstract:

We describe the Aspect Markup Language (AML), an XML-based AOP language for programming aspects. AML separates the binding instructions, written in XML, from the executable aspect code, written in a regular programming language. This separation by itself has some advantages, namely for testing. But the main goal of AML is to provide a highly extensible AOP platform, with which programmers can easily define their own constructs using well-known plugin techniques. This novel feature enables the development of AOP toolkits that target domain-specific crosscutting concerns. To demonstrate the feasibility of our approach, we have implemented AML for Java, along with a corresponding aspect weaver, jamlc. Jamlc weaves aspects written in AML and Java components into target bytecode. We present examples and show how to write aspect plugins.

The Aspect Markup Language and its Support of Aspect Plugins

Cristina Videira Lopes and Trung Chi Ngo
Institute for Software Research and Bren School of Information and Computer Sciences
Department of Informatics
University of California, Irvine
{lopes, trungcn} @ ics.uci.edu
ISR Technical Report UCI-ISR-04-8

ABSTRACT
We describe the Aspect Markup Language (AML), an XML-based AOP language for programming aspects. AML separates the binding instructions, written in XML, from the executable aspect code, written in a regular programming language. This separation by itself has some advantages, namely for testing. But the main goal of AML is to provide a highly extensible AOP platform, with which programmers can easily define their own constructs using well-known plugin techniques. This novel feature enables the development of AOP toolkits that target domain-specific crosscutting concerns.

To demonstrate the feasibility of our approach, we have implemented AML for Java, along with a corresponding aspect weaver, jamlc. Jamlc weaves aspects written in AML and Java components into target bytecode. We present examples and show how to write aspect plugins.

Categories and Subject Descriptors
D.1.1 [**Programming Techniques**]: General. D.3.3 [**Programming Languages**]: Language Constructs and Features – *data types and structures, modules, packages, patterns.* D.3.4 [**Processors**]: *Incremental compilers, preprocessors*.

General Terms
Programming languages, design.

Keywords
Aspect-Oriented Language Design, Extensible Languages, XML, Domain-Specific Languages.

1. INTRODUCTION
For quite a long time, modularization and separation of concerns have been known as the key for managing complexity in software systems [24]. Based on these two concepts, the software community has developed numerous programming techniques, such as procedure-oriented programming (POP), object-oriented programming (OOP), and, more recently, aspect-oriented programming (AOP). OOP's functional decomposition approach, with which designers break down a system into units of behavior, has been a successful programming technique; it fails, however, to handle special concerns that *crosscut* the system [10], [15]. Several studies have shown how crosscutting results in tangled code (e.g. [19]). This code tangling problem is not necessarily the result of the negligence of its authors, but the lack of an adequate mechanism in programming languages to deal with such concerns. AOP has been introduced to better model these crosscutting concerns.

Over the last few years, numerous aspect-oriented technologies have come into existence, including AspectJ [16], Composition Filters [5], Hyper/J [20], and recently AspectWerkz [4] and JBossAOP [14]. AspectJ is generally considered as the reference language for AOP. It provides many advanced techniques to deal with crosscutting concerns. The detailed discussion of those features is beyond the scope of this paper; readers can refer to [3],[8],[16],[17] for more information.

AspectJ has two shortcomings. First, being an extension of Java, it invalidates many existing software engineering tools available for plain Java only. The complex syntax of AspectJ's extensions discourages third parties from developing supporting tools, because they must build parsers in order to have access to the structure of the source code in AspectJ programs. Second, being a general-purpose aspect language is both good and bad. It's good because it supports the modeling and programming of a wide variety of crosscutting concerns. But it's bad because programmers must do so using a generic and relatively low-level model of crosscutting. The following is a brief discussion of these two issues.

1.1 Language Extensions
While some groups have the brain and financial power to support AspectJ's extensions, many don't, and support for AOP's useful

concepts have followed alternative solutions. For example, AspectWerkz has introduced an alternative for specifying aspects that avoids extensions altogether by taking advantage of XML [28]. In AspectWerkz, advice is implemented in regular Java classes and the binding instructions are specified using an XML-based aspect language. Although it is too early to conclude on the outcome of the friendly competition between AspectJ and AspectWerkz, the abundance of software engineering tools for Java and XML promises great opportunities for widespread adoption of approaches similar to AspectWerkz's.

1.2 General Purpose vs. Domain Specificity

The first steps towards formulating AOP have been taken within Domain-Specific Language (DSL) design [12],[20],[21],[22]. In AspectJ, those early ideas were expanded and generalized. AspectJ, AspectWerkz and most other aspect-oriented technologies are general-purpose. However, many people, including the authors, believe that the road to solving the problem of unnecessary software complexity involves letting the domain experts have direct control over the software by providing them appropriate expression mechanisms for their domain. This has been the drive behind DSLs. A DSL is a small, usually declarative, language that offers expressive power in the form of programming constructs focused on a particular problem domain. This ideal of DSLs, however, must take the following observation into consideration:

Software systems for "domains" are composed of more than one expertise. For example, a typical software application in Bioinformatics requires knowledge of how to analyze DNA sequences (Biology), how to access remote databases (Web/DB programming), how to define appropriate algorithms (Theoretical CS/Statistics), and how to efficiently implement those algorithms in large clusters of computers (Operating Systems) – see [6],[25]. A domain-specific language may fall prey to addressing all of that, becoming too big, too general, too stiff and/or too complex, therefore defeating its purpose.

1.3 Contributions of This Work

Our work provides an extensible aspect model and language that: (1) is applicable to any modern OOP programming language, (2) supports advanced AOP techniques, (3) leverages on existing software engineering tools, and (4) provides appropriate, and lightweight, support for domain-specific concerns. To achieve this, we started with AspectJ's join point model and built on AspectWerkz's XML-based representation of aspect definitions. We support domain-specificity by leveraging on XML's extensibility, providing an open implementation of the aspect weaver, with which programmers can extend the language using standard plugin techniques.

This paper describes our Aspect Markup Language (AML). The rest of this paper is organized as follows. Section 2 explains the aspect markup language and shows a couple of aspect plugins. Section 3 describes JAML, a mapping of AML for Java. Section 4 discusses related work. Finally, section 5 concludes with a discussion of our findings and future research directions.

2. THE ASPECT MARKUP LANGUAGE

The main philosophy behind the construction of the aspect-markup language (AML) is: 1) to not reinvent the wheel and 2) to provide a language-independent framework for specifying aspects in XML. We exploit existing AOP concepts as much as possible, and adopted AspectJ as the foundation for developing AML.

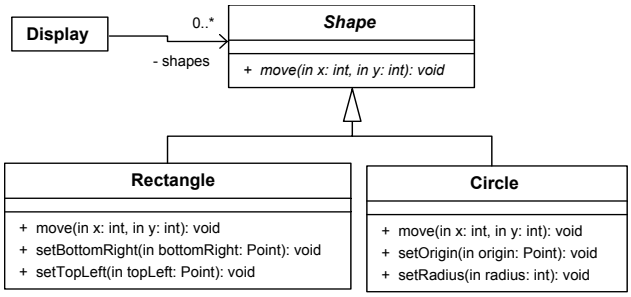

Figure 1. UML class model of the figure editor

In order to maintain its generality, the AML approach provides hints for language mappings without specifying anything that is platform, base language, or implementation specific. Those hints focus on specifying the structure of an aspect program and the architecture of an aspect weaver.

In this section, we explain AML and its components in general. We also mix examples throughout this section, as well as the rest of the paper, to illustrate the text better. Those examples are related to a *figure editor* program. Figure editor is a GUI program that is similar to the one shown in [16]. It provides a drawing canvas for users to draw geometric shapes. Two kinds of shape implemented in this program are Rectangle and Circle. Figure 1 shows the UML class model of the example that we use throughout the paper.

2.1 The Basics

As shown in Figure 2, an AML-based aspect program consists of four elements: core modules, aspect modules, plugin modules, and aspect definitions.

Core modules: Parts of the system that implement basic concerns, the primary functionality of the system. These modules are developed in the base language in the traditional OOP fashion.

Aspectual modules: Parts that implement subsidiary concerns affecting a crosscut of the system. These modules are developed in the base language, and are related to modules provided by AML (e.g. by inheritance).

Aspect bindings: XML-based binding specifications, written in some language mapping of AML. They provide binding instructions that determine how core and aspectual modules are unambiguously composed to produce the final behavior. In general, an aspect binding file contains a mixture of predefined XML elements called *core elements* and user-defined XML elements called *custom elements*.

Plugin modules: Parts of the program that provide specifications for custom elements used in aspect binding modules. These modules are developed in the base language with the aid of modules provided by the compiler framework.

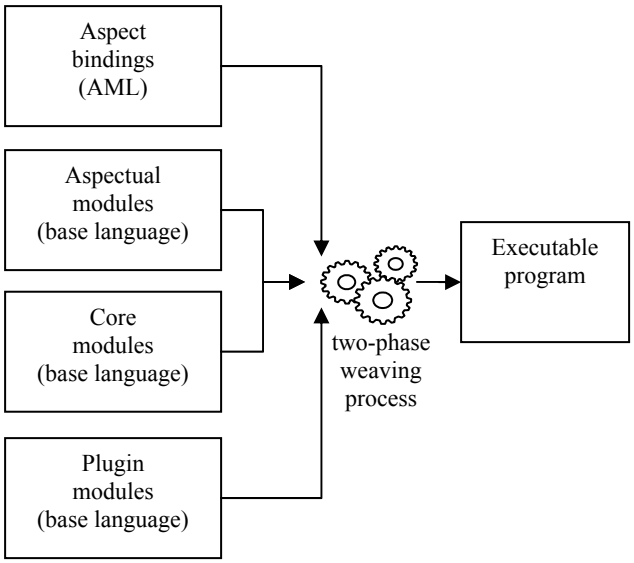

Figure 2. AML approach

Core elements, such as `<aspect>`, `<pointcut>`, `<interceptor>`, `<advice>`, and `<introduction>`, form the basis of AML. They are used to capture crosscutting elements of the system. Each *aspect* consists of *pointcut*, *interceptor*, and/or *introduction* definitions. A pointcut definition specifies a set of well-defined points called *join points* in the core modules based on signature patterns. An interceptor definition provides a means to implement crosscutting behaviors in AML. It specifies the name of an interceptor component among aspectual modules, and the methods of that interceptor that are invoked when certain join points are reached. An interceptor is a component that provides the implementation for one or more advice. An introduction definition provides a means of implementing static crosscutting concerns. It specifies what and how the static type structure of components in the core modules is extended with interfaces and corresponding implementation in the aspectual modules.

Besides core elements, programmers can use custom elements for specifying aspects. Custom elements are programmer-defined constructs whose syntax and semantics are given by plugin modules. For each of its custom elements, a plugin module provides the following information to the aspect weaver: 1) a combination of core elements that is equivalent to the custom element and 2) additional tasks that the aspect weaver needs to perform during the weaving process. Typical tasks are syntactic and semantic validations.

We add a new crosscutting requirement to the figure editor program to show an example of AML-based aspect program. This requirement is about implementing a new feature called *undo,* in which any operation that changes the visual representation of a shape needs to be recorded so that users can undo the change if they want to. For the purpose of the example, we assume that the historical record applies to all methods shown in Figure 1. The AOP solution for this is to implement an advice that runs before the execution of those methods and saves original information about the visual representation of the shape that is going to change.

```
<aml>

 <aspect id="History">

  <pointcut id="update">
   <or>
    <pointcut type="method-call"
       pattern="* *.move(..)"/>
    <pointcut type="method-call"
       pattern="* Rec*.setTopLeft(..)"/>
    <pointcut type="method-call"
       pattern="* Rec*.setBottomRight(..)"/>
    <pointcut type="method-call"
       pattern="* Circle.setOrigin(..)"/>
    <pointcut type="method-call"
       pattern="* Circle.setRadius(..)"/>
   </or>
  </pointcut>

  <interceptor
     class = "HistoryInterceptor">
   <advice
      type = "after"
      pointcut-refid = "update"
      interceptor-method  =
          "public void beforeUpdate()"/>
  </interceptor>

 </aspect>

</aml>
```

Figure 3. Aspect bindings for a History aspect

Figure 3 shows how the above solution can be implemented in JAML. The first section of the snippet defines the `History` aspect that consists of a composite pointcut and an interceptor. The composite pointcut picks out all calls to all methods. The interceptor definition has a nested advice definition that specifies that the `beforeUpdate()` method of `HistoryInterceptor` class (Figure 4) is an advice for the `update` pointcut. This method is invoked just before any `update` point is executed.

In this example, the core modules consist of all classes that implement basic functionality related to the drawing canvas, including those shown in Figure 1. The aspectual modules consist of code that handles crosscutting requirements, in this case the `HistoryInterceptor` class shown in Figure 4. The aspect bindings file shown in Figure 3 provides binding instructions between the `HistoryUpdate` aspectual module and the core modules `Shape`, `Rectangle` and `Circle`.

```
import edu.uci.jaml.lang.Interceptor;

public class HistoryInterceptor
  extends Interceptor {

  public void beforeUpdate() {

    // the code for saving original
    // information about the target
    // shape goes here

  }
}
```

Figure 4. Aspect module

```
<aspect enabled="yes">

 <pointcut id="myMethods">
  <or>
   <pointcut type="method-call"
      pattern="* Shape.*(..)"/>
   <pointcut type="method-call"
      pattern="* Circle.*(..)"/>
   <pointcut type="method-call"
      pattern="* Rectangle.*(..)"/>
  </or>

 </pointcut>

 <interceptor
   class="TraceInterceptor">

  <parameters>
    <!-- initialize the interceptor with
       logFile field = 'trace.log' -->
    <param id="logFile" value="trace.log"/>
  </parameters>

  <advice type="before"
     pointcut-refid="myMethods"
     interceptor-method=
        "void beforeMyMethods()"/>

  <advice type="after"
     pointcut-refid="myMethods"
     interceptor-method=
        "void afterMyMethods()"/>

 </interceptor>

</aspect>
```

Figure 5. Tracing aspect with core elements

The JAML code that specifies the `History` aspect is quite verbose; however, it does not necessarily have to be directly written by the user. It can be generated by a smart IDE through user-friendly interfaces.

2.2 ASPECT-SPECIFIC PLUGINS

AML enables programmers to extend the aspect language by defining their own constructs. This novel feature enables the development of AOP toolkits that target domain-specific crosscutting concerns. In this section, we present two examples of how to use plugins, to illustrate this feature and to show benefits gained from using it. In section 4 explains what is involved in developing language extensions through plugins.

2.2.1 A plugin for tracing

Before presenting the use of the tracing plugin, we describe how to do tracing in plain JAML. Figure 5 illustrates this. The code in Figure 5 defines a pointcut consisting of method calls to all methods of Shape, Circle and Rectangle objects, and binds two methods of the aspectual module called TraceInterceptor to advice actions.

Figure 6 shows the JAML code that traces all calls to any method of `Shape`, `Circle`, and `Rectangle` classes, using a tracing-specific plugin we have developed. In order to use the tracing aspect, programmers simply specify the list of classes whose methods need to be traced, and the output file name.

```
<!-- traces all calls to any method of Shape,
     Circle, and Rectangle classes, stores
     output in a file named "trace.log" -->
        -->
<debug:tracingAspect
  enabled="yes"
  logFile="trace.log">

 <debug:methods>
  <debug:entry class="Shape"/>
  <debug:entry class="Circle"/>
  <debug:entry class="Rectangle"/>
 </debug:methods>

</debug:tracingAspect>
```

Figure 6. Tracing aspect with custom elements

Although it is possible to specify the tracing aspect using only standard AOP constructs (Figure 5), the code with custom elements (Figure 6) shows significant advantages, as it provides better support for the simple task of tracing all methods of certain classes. For aspect writers, especially for non-experts of AOP, it is easier to think of the problem in terms of classes whose methods need to be traced, rather than to think of a complex composition of pointcuts, interceptors, and advice. Moreover, compared with its plain JAML counterpart, the code with custom elements is more concise and self-descriptive. Without exposing any mechanics behind the scenes, it still provides sufficient information for readers to make correct inferences about the purpose of the code.

2.2.2 A plugin for GoF design patterns

Gang-of-Four (GoF) design patterns [8] provide flexible design solutions for common software development problems. They are presented as a means of formalizing and distributing knowledge of good designs in Object-Oriented languages.

Although commonly known implementations of GoF patterns are written in OOP languages, recent work [10] has shown that AOP implementations can yield better code locality, reusability, composability, and (un)pluggability. [10] also proposes a set of AspectJ abstract aspects for implementing patterns. The construction of this aspect library relies on the observation that most design patterns have some parts that are common to all potential instantiations and others that are specific to each instantiation. As a result, each abstract aspect in the library encapsulates the common parts of a pattern and provides instructions/plug points for programmers to implement the instantiation-specific parts in the concrete aspect.

Based on observations and design suggestions discussed in [10], we developed a plugin that provides a mini language for expressing GoF design patterns. Figure 7 shows an example of how programmers can use the language to define an instantiation of Singleton and Observer patterns. The first part in the snippet shows how programmers can make the `Display` class become a singleton object. The second part shows an instantiation of `Observer` pattern, in which the `Rectangle` class is a subject to be observed and the `Display` class is an observer.

```
<gof:patterns>
 <!-- apply singleton pattern to the
      Display class but not its sub classes -->
 <gof:singletonPattern>
  <gof:class name="Display"
     excludeSubclasses="yes"/>
 </gof:singletonPattern>


  <!-- apply observer pattern to Rectangle
       and Display classes -->
 <gof:observerPattern>

   <!-- make Rectangle class
       a subject (observable) -->
  <gof:subject class="Rectangle">

    <!-- define a list of methods that
        change the states of the subject -->
    <gof:mutators>
     <gof:method signature="* move(..)"/>
     <gof:method signature="* set*(..)"/>
    </gof:mutators>

  </gof:subject>

  <gof:observer
     class="Display"
     implementation = "DisplayObserverImpl"/>
 </gof:observerPattern>

</gof:patterns>
```

Figure 7. Example of Singleton and Observer patterns

The mechanics behinds our implementation of GoF design patterns is relatively similar to the AspectJ implementation discussed in [10]. For instance, in the case of the Singleton pattern, during compiling time we generate AspectJ code that turns original constructors into factory methods using an `around` advice that returns the unique object on all constructor calls. In addition, our implementation also provides facilities to exclude subclasses from the Singleton protection. As shown in Figure 7, programmers can turn on or off singleton protection on subclasses simply by setting `excludeSubclasses` attribute of `<gof:class>` element to either `"yes"` or `"no"` respectively. If the attribute is set to `"yes"`, we generate an AspectJ pointcut exclude all subclasses from the Singleton protection. The AspectJ implementation, in contrast, requires programmers to specify that pointcut directly.

Compared to the AspectJ implementation of GoF design patterns proposed in [10], our implementation shows improvements in terms of the model that is presented to the end-programmer. The language of our patterns plugin is designed in such a way that it is accessible to programmers who have an adequate knowledge of object-oriented programming and design patterns, but that may know very little of AOP.

2.3 Aspect plugins vs. abstract aspects

There are many similarities between our aspect plugins and abstract aspects. In both cases, the goal is to provide the end-programmer with a predefined package for implementing certain crosscutting concerns, while allowing her to specify certain parameters of operation of those packages. The similarities here parallel those between library functions using a general-purpose language and solutions based on domain-specific languages. That is, in fact, what we are supporting – lightweight aspect-specific languages. We believe these languages play an important role in managing software complexity. An abstract aspect that is concretized in some application requires a deep understanding of the underlying join point model; in our approach, that requirement is removed from the end-programmer, and, instead, falls under the responsibility of the specialized programmer who produces the aspect-specific plugins.

2.4 Two-phase Weaving Process

Due to the nature of mixing core and custom elements, the AML-based aspect weaver needs to adopt the two-phase weaving (TPW) process. As suggested by its name, the weaving process has two distinct phases: *pre-process* and *actual weaving*.

Pre-process phase: During the pre-process phase, the aspect weaver parses all aspect definition files, builds an in-memory intermediate abstract-syntax structure (AST) that consists of nodes representing core elements and/or custom elements. Then, syntactic and semantic validations need to be performed on each node of the AST. Finally, for each node representing a custom element, the weaver performs all associated tasks and then replaces the node by its equivalent combinations of nodes representing core elements. The result of this process is an AST consisting of only nodes that represent core elements.

Actual weaving: In this phase the aspect weaver performs traditional aspect weaving to build the executable, given core modules, aspectual modules, the AST and additional artifacts (e.g. source files, binary components, and data files) resulting from the pre-process phase. The implementation of this phase has been a subject of intensive research since the birth of AOP.

To give an intuition on how the TPW works, it suffices to say that, referring to the tracing example, the first phase converts the code in Figure 6 to code similar to that of Figure 5, i.e. the custom elements are translated into primitive elements. The weaver takes it from there.

3. JAML –AML MAPPING FOR JAVA

Any AO language needs to have three critical elements: a join point model, a means of identifying join points, and a means of effecting implementation at join points [16]. AML describes a set of basic XML elements and their attributes that all language-specific mappings needs to support and extend. In order to demonstrate our approach, we developed JAML, a mapping of AML for Java.

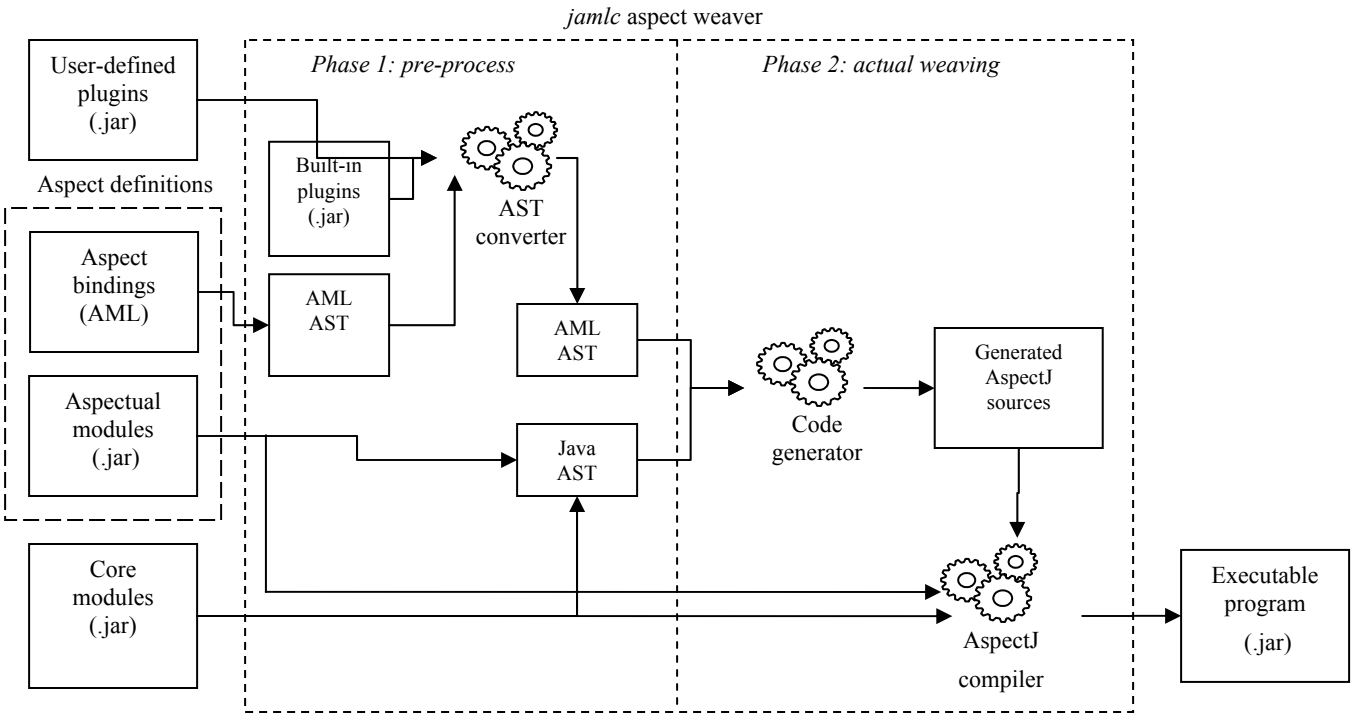

Figure 8. JAML's weaving process

3.1 The join point model

JAML's join point model is derived directly from the dynamic join point model of AspectJ version 1.1. A join point is a well-defined point in the execution flow of a program, and a pointcut is a set of join points that define a crosscut of the system where crosscutting behaviors take place. There are two types of pointcut, namely *primitive* and *composite pointcuts*. In AspectJ, a primitive pointcut picks out all join points of a *kind* (e.g. call, execution, get, and set) with the constraints defined in terms of textual patterns. In JAML, a primitive pointcut can be specified in the same fashion. For example, the following snippet shows how to specify a primitive pointcut referring to all join points at which `setX()` method of `Point` class is called.

AspectJ:

```
pointcut p :
  call(void Point.setX());
```

JAML:

```
<pointcut
  id= "p"
  type = "method-call"
  pattern = "void Point.setX()"/>
```

There are 16 types of primitive join point in AspectJ [3], and the current version of JAML supports all of them, except *if* and *adviceexecution* join points.

In AspectJ, boolean operators AND, OR, and NOT (denoted `&&`, `||`, and `!` respectively) can be used to combine other pointcuts into a complex composition of join points called *composite pointcut*. For example, the pointcut composed by `within(Line) && (call(void Point.setX() || call(void Point.setY()))` picks all join points in the implementation of type `Line` and at which `setX()` and `setY()` methods of `Point` class are called. This feature is useful to specify a complex set of join points that is a composition of intersections, unions, and/or complements of other sets of join points.

The above example of a composite pointcut is expressed in JAML as follows:

```
<pointcut id="p">
 <and>
  <pointcut
     type = "within"
     pattern = "Line"/>
   <or>
    <pointcut
       type = "method-call"
       pattern = "void Point.setX()"/>
    <pointcut
       type = "method-call"
       pattern = "void Point.setY()"/>
   </or>
  </and>
</pointcut>
```

3.2 Dynamic crosscutting

In AspectJ, advice declarations are used to define additional code that needs to be executed when certain join points are reached. This feature, which provides a means of intercepting the execution a program, is also referred as dynamic crosscutting technique.

The current aspect model of AspectJ has five types of advice, and JAML supports all of them. In JAML, advice definitions are specified using `<advice>` elements, nested in an `<interceptor>` element. `class` attribute of `<interceptor>`

element provides clues for the aspect weaver to find an implementation of the interceptor in the given aspect modules. Each nested `<advice>` element specifies a pointcut that needs crosscutting actions and a method of the interceptor that provides the actual advice implementation.

An interceptor is implemented as a regular Java class that extends from `edu.uci.jaml.lang.Interceptor` class provided by the framework. Inherited from its super class, any interceptor has access to a protected field called `thisJoinPoint`. This special field provides reflective access to the context published by the currently executing join point, and through which users can perform advanced AOP techniques.

3.3 Static crosscutting

Besides dynamic crosscutting, AspectJ also provides the concept of introductions (inter-type declarations) that are used to extend the static type hierarchy of a program. This feature is also documented as the static crosscutting mechanism. It allows users to insert additional class members into existing types, including constructors, methods, and fields.

In JAML, `<introduction>` element provides a similar mechanism to express static crosscutting concerns. It allows users to insert new abstractions to existing Java classes. The current implementation of JAML supports method and constructor introductions, but not field introductions.

3.4 Implementation

The current aspect weaver for JAML is implemented as a command-line tool called `jamlc`. It runs on top of `ajc`, AspectJ's aspect weaver. During weaving time, `jamlc` translates AML-based aspect definitions to equivalent AspectJ source code and then delegates the actual weaving phase to `ajc`. The `jamlc` program accepts inputs consisting of aspect bindings in XML and already compiled core, aspectual, and plugin modules in .jar format. The `jamlc` tool performs standard aspect weaving in which core and aspect modules are processed properly to compose the desired total system operation [15].

During the pre-process phase, the first phase of the TPW, `jamlc` parses, performs syntactic checking, and then constructs the in-memory model of the inputs that consist of an internal representation of aspect bindings as abstract-syntax tree (AST) and a static type hierarchy of the core and aspectual modules. For each node representing a custom element in the AST tree, the AST Converter component performs the associated tasks and then replaces the node with its equivalent combination of nodes representing core elements. Semantic analysis is performed, to a certain degree, on the given model before sending it to the second phase, the actual weaving phase.

In order to make a prototype of the aspect weaver available early, so that we can perform experiments on our approach before implementing a full-blown front-end aspect compiler, we decided to exploit the power of the AspectJ compiler. During second phase of the TWP approach, the code generator component of the `jamlc` program generates equivalent AspectJ source files and then internally delegates the weaving task to the AspectJ's aspect weaver.

Besides helping to shorten the development time significantly, this source-to-source translation approach also keeps the implementation of the aspect weaver for JAML short and concise. Nevertheless, flexibility is lost. The current aspect model of JAML is confined by that of AspectJ.

3.5 Example

In this section, we show the generated AspectJ source of the aspect definition of the `History` aspect discussed earlier to elaborate on the mechanism behind the `jamlc` aspect weaver.

The snippet in Figure 9 shows the generated AspectJ source code for the `History` aspect shown in Figure 3 during the weaving process. The structure of the generated AspectJ code is quite simple and almost identical to that of the XML representation. The generated code defines an AspectJ aspect that has a composite pointcut, an interceptor defined as a protected field of the aspect, and a *before* advice. When the advice body is executed, it propagates the current join point's context (i.e., the `thisJoinPoint` object) to the interceptor object and then invokes the `beforeUpdate()` method of the interceptor to the actual advice implementation.

4. WRITING PLUGINS FOR JAML

One important design feature of the `jamlc` is that it maintains a loose coupling between the parser and the custom elements. This enables programmers to write aspect plugins easily. In addition, new plugins can be added to the framework without having to recompile the implementation of the aspect weaver. In doing so, we make use of well-known introspection techniques to access interfaces exposed by the plugin modules.

```
import edu.uci.jaml.lang.JoinPoint;
import HistoryInterceptor;

public aspect Aspect$History {

 protected pointcut pointcut$needUpdate():
 (
   call(* *.move(..)) ||
   call(* Rec*.setTopLeft(..)) ||
   call(* Rec*.setBottomRight(..)) ||
   call(* Circle.setOrigin(..)) ||
   call(* Circle.setRadius(..))
 );

  protected HistoryInterceptor
      interceptor$1 = new
         HistoryInterceptor();

  before() : pointcut$needUpdate() {

    JoinPoint savedJoinPoint =
       interceptor$1.getCurrentJoinPoint();
    interceptor$1.setCurrentJoinPoint(
       new JoinPoint(thisJoinPoint));

    interceptor$1.beforeNeedUpdate();

    interceptor$1.setCurrentJoinPoint(
       savedJoinPoint);
  }
}
```

Figure 9. Generated AspectJ source of History aspect

```
<debug:methods>
 <debug:entry
    class="Shape"/>
</debug:methods>
```

Figure 10. Simple composition of <debug:methods> and <debug:entry> elements.

Writing a custom element for JAML is not much different from writing a custom task for Apache Ant build system. In this section, we review that process, highlighting the differences.

JAML provides a simple plug point at which programmers need to write code that responds to certain predefined events occurring during the weaving process. The current implementation of `jamlc` requires a custom element to implement the `edu.uci.jaml.lang.util.CustomElement` interface whose specification serves as an formal contract for the parser to recognize the element.

4.1 Life Cycle of a Custom Element

To gain an understanding of how the `jamlc` aspect weaver interacts with a custom element, we need to examine its life cycle. The execution of any custom element passes through four distinct phases, *creation*, *setting attributes*, *adding nested elements*, and *evaluation*. Before finishing its life cycle, a custom element is expected to return a list of objects to its parent. For a top-most custom element, i.e. a custom element that is nested directly in a core element, the returned list is the equivalent combination of AST nodes representing only core elements.

Creation phase: While parsing an aspect bindings file, the embedded parser recognizes the use of any element that is not core element. Based on the fully qualified name (prefix + local element name, such as `<debug:tracingAspect>`) of the element, the parser then tries to resolve the corresponding implementation from the input plugin modules. A new instance of the implementation, called custom element bean, is created if one is found. Otherwise, the parser will report a build exception. The no-argument default constructor of the custom element bean will be executed during this phase.

Setting attribute phase: In this phase, the parser will perform syntactic validation on the set of attribute values specified in the aspect definition file against its definition returned by `getAttributeDefinitions()` method of the custom element bean. If the attribute values are all valid, the parser makes use of introspection techniques to propagate those values to their corresponding attributes in the bean. In order to do so, the bean must include appropriately named set methods. For example, if the custom element has a String attribute named *id*, its bean must have a public method whose signature is "`public void setId(String)`".

Adding nested elements phase: In this phase, the parser will parse each nested element and add the resulted list of objects to the current element through appropriately named add method. For example, if the resulted list contains an object that is typed of `XmlDebugEntryElement`, the custom element bean must have a public method whose signature is "`public void addXmlDebugEntryElement(Object)`".

```
public class XmlTraceEntryElement
                  implements CustomElement {

// the class attribute
private String attClass;

public AttributeDefinitionList
              getAttributeDefinitions() {
  // return attribute list
  // of definition for the pattern attribute
  return new AttributeDefinitionList(
            new AttributeDefinition(
               "class", //attribute name
               true,    //required attribute
               false,
               null)});
}

public String getName() {return "entry";}

// setter for pattern attribute
public void setClass(String value) {
  this.attClass = value;
}

public String getClazz() {
  return attClass;
}

public List execute() {
  // return itself to its parent
  List result = new LinkedList();
  result.add(this);
  return result;
}
}
```

Figure 11. Implementation for <debug:entry> element

Evaluation phase: After parsing all attributes and nested elements, the parser will call `execute()` method of the custom element to execute tasks associated with the current element. The method returns a list of objects that will be added as nested objects to the parent element. As mentioned earlier, if the element is nested directly in a core element, the return list is the equivalent combination of nodes representing only core elements.

4.2 Example

In order to define a new custom element, programmers can simply implement a no-argument constructor, *set* methods for all attributes, *add* methods for its nested elements, and a few other methods, including those required by the `CustomElement` interface.

In order to illustrate the above process, we describe in detail the implementation for `<debug:entry>` and `<debug:methods>` custom elements discussed earlier. Figure 10 shows an example composition of those elements, which is a simplified version of the snippet shown in Figure 6. The semantics of this composition is to define a composite pointcut consisting of only pointcuts that are calls to a method of `Shape` class.

```
public class XmlTraceMethodsElement
                implements CustomElement {

// list of nested entries
private List nestedTraceEntries =
                            new LinkedList();

public AttributeDefinitionList
             getAttributeDefinitions() {
  // return empty attribute list
  // it means the element has no attributes
  return new AttributeDefinitionList(
              new AttributeDefinition[] {
             });
}

public String getName() {return "methods";}

// adder for nested element that is
// XmlTraceEntryElement type
public void addXmlTraceEntryElement(
         XmlTraceEntryElement nestedEntry){

 // add the nested trace entry to the list
 nestedTraceEntries.add(nestedEntry);
}


public List execute() {

 // build composite pointcut of its entries
 AstPointcut astPointcut =
  new AstCompositePointcut(
   null,       // unknown parent aspect
   null,       // anonymous pointcut
   AstCompositePointcut.OperatorTypeEnum.OR);

 // for each nested entry, create a method
 // call pointcut with its pattern is
 // that of the entry, and add the resulted
 // pointcut to the composite pointcut
 Iterator it = nestedTraceEntries.iterator();
 while (it.hasNext()) {
  XmlTraceEntryElement e =
          (XmlTraceEntryElement) it.next();

  //create a method-call, and make the advice
  //reference to the newly created pointcut

  AstPointcut p = new AstMethodCallPointcut(
      null,     // unknown parent aspect
      null,     // anonymous pointcut
      "* " + e.getClazz() + "*.(..)");

  astPointcut.addChildPointcut(p);

  // return the poincut to its parent
  List result = new LinkedList();
  result.add(astPointcut);
  return result;
}
}
```

Figure 12. Implementation for <debug:methods>

Figure 11 shows the simplified version of the `XmlTraceEntryElement` class which implements desired functionality of `<debug:entry>` custom element. In this snippet, `getName()` function of the `XmlTraceEntryElement` class returns the "*entry*" string. It indicates that `XmlTraceEntryElement` provides implementation for a custom element whose local name is *entry*. The returned value of

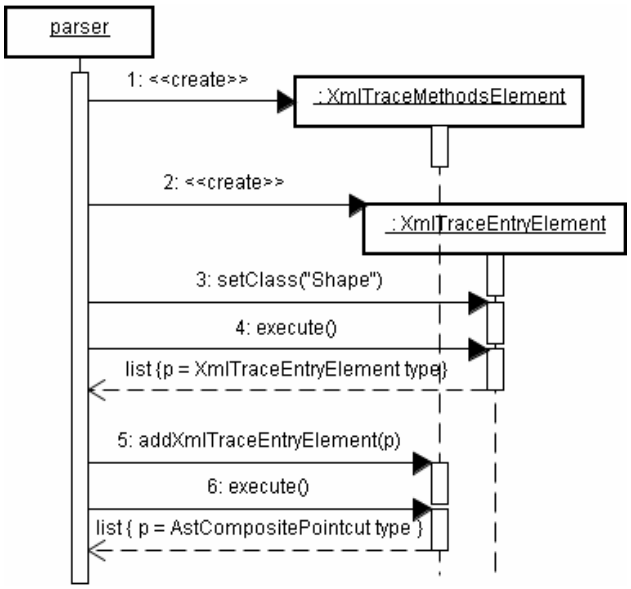

Figure 13. Sequence diagram showing how the parser interact with custom element classes

`getAttributeDefinitions()` method is a list of attribute definitions. It shows that the current custom element has only one required attribute named *class*. The parser can set the value of the pattern attribute through its set method whose signature is "`public void setClass(String)`". At the last phase of its cycle, `execute()` method returns a list that consists of a reference to the current instance of `XmlTraceEntryElement` class. Noting is that the parser will later add this reference to an instance of `XmlTraceMethodsElement` as a nested object.

Figure 12 shows the simplified version of `XmlTraceMethodsElement` class which implements desired functionality of `<debug:methods>` custom element. The structure of `XmlTraceMethodsElement` class is the same as that of `XmlTraceEntryElement` class. The returned value of `getName()` function shows that `XmlTraceEntryElement` class provides implementation for a custom element whose name is *methods*. `getAttributeDefinitions()` function returns an empty list of attribute definitions, indicating that the element does not accept any attribute. The add method whose signature is "`public void addXmlTraceEntryElement(..)`" indicates that the element accepts nested objects that are `XmlTraceEntryElement` type. At the last phase of its cycle, `execute()` method returns a composite pointcut which is a set of join points at which methods of classes specified in `XmlTraceEntryElement` objects are called. `AstPointcut`, `AstCompositePointcut`, and `AstMethodCallPointcut` are AST classes representing the `<pointcut>` core element. These classes are parts of `jamlc` compiler.

Following the flow in the sequence diagram shown in Figure 13 helps to visualize the interactions between the parser and the implementation of custom elements. Specifically, this sequence diagram describes how the parser interacts with `XmlTraceEntryElement` and `XmlTraceMethodsElement` classes while parsing the snippet shown in Figure 10.

The diagram shows that the parser will first create new instance of `XmlTraceMethodsElement` class to handle `<debug:methods>`

custom element (step 1). After recognizing another custom element named `<debug:entry>`, the parser creates a new instance of `XmlTraceEntryElement` class (step 2). The parser calls `setPattern()` method of the newly created instance of `XmlTraceEntryElement` class to set its pattern attribute to "`* Shape.*(..)`" (step 3). `execute()` method of the `XmlTraceEntryElement` object is called. The returned value of this call is a list that consists of only a reference to the `XmlTraceEntryElement` object (step 4). The parser will invoke `addXmlTraceEntryElement()` method of the current `XmlTraceMethodsElement` object to add the reference to the `XmlTraceEntryElement` resulted in step 4 as a nested object (step 5). Finally, the parser invokes `execute()` method of the `XmlTraceMethodsElement` object to build the desired composite pointcut (step 6).

5. RELATED WORK

This work shares similarities with JBoss AOP [14], AspectWerkz [4], and Weaver.NET [18] in terms of the use of XML to specify aspects. None of these, however, has focused on extending their aspect languages. We briefly describe their strengths.

JBoss AOP, developed by JBoss Inc., is a general-purpose AOP framework that targets commercial Java/J2EE applications. Besides the ability to define crosscutting behaviors, adopting JBoss AOP provides other benefits that come from its tight integration with the JBoss application server. For instance, the JBoss server comes with a prepackaged set of system-level aspects such as caching, security, transaction, and asynchronous communications.

AspectWerkz is a lightweight framework targeted towards dynamic aspect-oriented programming for Java. AspectWerkz allows users to define aspects through either XML or JavaDoc tags. The aspect model of AspectWerkz is also derived directly from AspectJ. Current work on AspectWerkz focuses on supporting dynamic AOP, including the ability to add, remove, and reorder of advices as well as swaps the implementation of mixins (i.e., introductions) at runtime.

Weaver.NET demonstrates the feasibility of constructing a language-independent AOP platform for Microsoft.NET framework. The idea behind this work is to exploit the language interoperability characteristic of the Common Language Infrastructure (CLI) designed for the .NET platform. In this approach, programmers can choose any language (e.g., C#, VB.NET, and J#) that targets the .NET platform to implement aspect behaviors.

Besides the goal of providing an alternative mechanism for specifying aspects, our work on AML goes beyond the separation between aspect code and aspect bindings; our work is a step towards extensible AOP platforms. In JAML, programmers can easily add new constructs to the aspect language through implementing plugin components. This novel feature promotes the development of AOP toolkits that provide first-class reusable aspects and/or domain-specific syntactic constructs.

In this respect, our work shows similarities with XAspects [26], and, more broadly, with all the work related to extensible languages and open compilers. XAspects, however, stayed away from XML. One of the goals of that work was to model structure-shy problems, and for that, better support was found in ANTLR [1].[1] We believe XML is a simple, but powerful language. Its wide deployment makes it a good vehicle for transforming the work in domain-specific languages from being limited to just a few experts to being accessible to a much larger community.

6. CONCLUSIONS AND FUTURE WORK

The aspect-markup language (AML) is a language-independent model for constructing XML-based AOP languages. AML provides hints for the language mappings without specifying things that are platform, base language, or implementation specific. In AML, an aspect-oriented program consists of three elements: core modules, aspectual modules and aspect bindings. Core and aspectual modules are developed in the base language, which is an existing OOP language (e.g., Java, C++, and C#), and aspect bindings are specified in AML. Further, it is possible to provide weaver plugins that support extensions of the base aspect language. These artifacts are processed in a two-phase weaving to produce the final executable program.

JAML is mapping of AML for Java. The current implementation of the aspect weaver for JAML runs on top of `ajc`, AspectJ's aspect weaver. During weaving time, it translates aspect definitions to equivalent AspectJ source code and then delegates the actual weaving phase to `ajc`.

From a programmer's perspective, the idea of separating aspect definitions from their actual implementations gives adopters more flexibility in terms of development tools. For example, in the case of JAML, both primary and crosscutting components are implemented as regular Java components. Thus, all existing software tools supporting Java (e.g., Java IDEs, refactoring, debugging, and testing tools) are still usable in JAML. Testing can also be made much easier, as the aspectual components are now regular Java classes.

Another contribution of this work is its support for extensible AOP languages. The motivation behind this approach is to promote the development of new AOP platforms that target both general-purpose and domain-specific concerns. The development of aspect plugins for JAML shows that it is possible to define new AOP languages based on a set of primitive AOP constructs. The combination of XML and plugin techniques can take AOP development to a new level, where a group of programmers with AOP expertise produces these aspect-specific plugins and a larger group without that expertise uses them.

The development of JAML is at early stages. In order to gain better acceptance, future work will involve providing better tool support. As mentioned earlier, it is tedious and error-prone for programmers to manually write aspect bindings in XML. We believe is it important to provide smart IDEs or GUI tools that offer context-sensitive assistance and code generation capabilities.

[1] Personal communication.

7. REFERENCES

[1] ANTLR http://www.antlr.org/

[2] Apache Ant, *Ant build system home page*, http://ant.apache.org/

[3] AspectJ, http://www.eclipse.org/eclipse/

[4] AspectWerzk, *AspectWerkz homepage*, http://aspectwerkz.codehaus.org/

[5] L. Bergmans and M. Aksit, *Composing multiple concerns using composition filters*, Communications of the ACM, Octorber 2001.

[6] BioPerl. http://bioperl.org/ ; tutorial at http://bioperl.org/Core/Latest/bptutorial.html

[7] J. Bonér, *What are the key issues for commercial AOP user – how does AspectWerkz address them*, AOSD, 2004.

[8] Gamma, E. et al. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Weslay, 1994.

[9] J. Gradecki and N. Lesiecki, *Mastering AspectJ*, Wiley, 2003.

[10] J. Hannemann and G. Kiczales. *Design Patterns Implementation in Java and AspectJ*. In Proceeedings of the 17th ACM conference on object-oriented programming, systems, languages, and applications, pages 161-173. ACM Press, 2002.

[11] W. Hürsch and C. Lopes, *Separation of Concerns*, Northeastern University technical report NU-CCS-95-03, 1995.

[12] Irwin, J., Loingtier, J.-M., Gilbert, J.R., Kiczales, G., Lamping, J., Mendhekar, A. and Shpeisman, T. 1997. *Aspect-oriented programming of sparse matrix code*. Scientific Computing in Object-Oriented Parallel Environments. First International Conference, ISCOPE 97. Proceedings. Springer-Verlag, 1997. p.249-56.

[13] JAML, *JAML home page*, http://www.ics.uci.edu/~trungcn/jaml

[14] JBoss Group, *JBoss aspect-oriented programming*, http://www.jboss.org/products/aop

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Lopes, Jean-M. Loingtier, J. Irwin., *Aspect-oriented programming*, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), June 1997.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, *Getting started with AspectJ*, Communication of the ACM, Volume 44, Issue 10, October 2001.

[17] I. Kiselev, *Aspect-oriented programming with AspectJ*, SAMS, 2003.

[18] D. Lafferty and V. Cahill, *Language-Independent Aspect-Oriented Programming*, Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.

[19] M. Lippert and C. Lopes, *A study on exception detection and handling using aspect-oriented programming*, In Proceedings of the 22nd International Conference of Software Engineering (ICSE'2000), Limmerick, Ireland. IEEE Computer Society. June 2000.

[20] Lopes, C. and Lieberherr, K. 1994. *Abstracting Function-to-Process Relations in Concurrent Object-Oriented Applications*. In proc. European Conference on Object-Oriented Programming (ECOOP'94). Springer-Verlag LNCS 821.

[21] Lopes, C., 1996. *Adaptive Parameter Passing*. In Proc. International Symposium on Object Technologies for Advanced Software (ISOTAS'96). Springer-Verlag LNCS n.1049. Japan, 1996.

[22] Mendhekar A., Kiczales G. and Lamping J. 1997. *RG: A Case-Study for Aspect-Oriented Programming*. Xerox Palo Alto Research Center Technical Report SPL97-009 P9710044. February 1997.

[23] H. Ossher and P.Tarr, *Multi-Dimensional Separation of Concerns and the Hyper-space Approach*, Proceedings of the Symposium on Software Architectures and Component Technology: The State of Art in Software Development, Kluwer 2000.

[24] D.L. Parnas. *On the criteria to be used in decomposing systems into modules*. In Communications of the ACM, 15(12):1053-1058, 1972.

[25] Sahni, H. *Cancer Bioinformatics Infrastructure Objects (caBIO)*. Poster in Bioinformatics Technology Conference, 2003. http://conferences.oreillynet.com/cs/bio2003/view/pos/34

[26] M. Shonle, K. Lieberherr and A. Shah. XAspects: *An extensible system for domain-specific aspect languages*. In proc. of OOPSLA'03, Anaheim, CA, October 2003.

[27] K. D. Volder, J. Brichau, K. Mens, and T. D'Hondt, *Logic meta-programming, a framework for domain-specific aspect programming languages*, http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf.

[28] W3C, *Extensible Markup Language (XML)*, http://www.w3.org/XML/