Modeling for Dynamic Aspect-Oriented Development

Farhana Eva Alam
Dept. of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada
fevaalam@cs.mun.ca

Joerg Evermann
Faculty of Business
Memorial University of Newfoundland
St. John's, NL, Canada
jevermann@mun.ca

Adrian Fiech
Dept. of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada
afiech@mun.ca

ABSTRACT

Aspect Oriented Software Development (AOSD) has its roots in the need to deal with requirements that cut across the primary modularization of a software system. On the programming level, mature, industrial-strength tools like the de-facto standard AspectJ exist. However, on the modeling level, there is as yet little support for AOSD. Building on previous work, this paper develops UML modeling support for dynamic AOSD, using standard UML extension mechanisms. We present a generic profile that allows existing UML tools to express AOSD models. We also provide automatic code generation into AspectS, an aspect extension to Smalltalk, and AspectML, an aspect oriented flavor of the ML language. Examples throughout the paper illustrate our approach.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features.

General Terms

Design, Languages, Theory.

Keywords

Aspect-oriented programming, aspect-oriented modeling, dynamic AOP.

1. INTRODUCTION

Aspect Oriented Software Development (AOSD) deals with requirements that cut across the primary modularization of a software system, e.g. logging, tracing, security, persistence. Initially developed as aspect-oriented programming (AOP) [1], it has led to a number of mature tools for different languages, such as AspectJ, the de-facto standard for AOP using Java [2]. Aspect-oriented extensions also exist for many other languages, including Smalltalk [3] and ML [4].

The core concepts of AOP are joinpoints, pointcuts, advice and aspects. Joinpoints are points in the execution of a software system. Pointcuts are sets of joinpoints selected by the AOP developer. Code can be attached to pointcuts. This code is specified in the form of advice. Related pointcuts and advice are modularized in an aspect. An aspect weaver automatically adds the advice code to the specified pointcuts. Different languages provide concepts beyond this core, such as static introductions in AspectJ, which allows the AOP developer to add class members and interface realizations [6]. In this paper, we focus on the core concepts only, as these are common across most AOP implementations. AOP approaches can be characterized as either static or dynamic. Static AOP, as implemented e.g. in AspectJ, requires the developer to specify all pointcuts, advice and aspects at compile time. Typically, a weaving compiler is used to add advice code to joinpoints. Dynamic AOP on the other hand, allows changes to aspects at run-time and a run-time weaver is used to add advice code to the selected joinpoints.

Aspect-oriented modeling (AOM) is the extension of AOSD into the upstream software design activities. It is increasingly important in the context of the OMG's model-driven development (MDD). Different AOM approaches are characterized by their genericity. Many of the existing AOM approaches are programming language specific and allow modeling on the PSM (platform specific model) level. While there are few AOM extensions to allow generic modeling on the PIM (platform independent model) level, they offer advantages as it increases the re-usability of the models, cooperation of developers with different language backgrounds and future-proofing of the software design.

This work-in-progress paper describes an AOM approach for platform-independent modeling for dynamic AOP. It builds on earlier work [5] that provided a platform-specific AOM approach for static AOP, specifically AspectJ. Our AOM approach is based on standard extensions to UML and is therefore usable in all CASE tools that support profiles and stereotypes. We provide code generation to AspectS [3] and AspectML [4] using XSLT (Extensible Stylesheet Language Transformations) translation on the UML XMI standard serialization format.

The paper proceeds as follows. Section 2 compares the features of different AOP extensions to Java, Smalltalk, and ML to ensure that our AOM approach covers important AOP features in a wide variety of languages. Section 3 introduces dynamic aspects using two examples to illustrate their advantages over static AOP. Section 4 illustrates our modeling approach. The paper concludes in Section 5.

2. JOIN POINT MODELS

While the core AOSD concepts are similar across most AOP implementations, there are subtle differences and unique features. This section discusses the join point model (JPM) of different AOP implementations. The JPM specifies which joinpoints in a software system's execution can be selected by pointcuts. To cover a wide variety of JPM features, we examine three languages: AspectJ – a static AOP approach, AspectS – a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
C3S2E-09 2009, May 19-21, Montreal [QC, CANADA]
Editor: B C. DESAI, Copyright ©2009 ACM 978-1-60558-401-0/09/05 $5.00

dynamic, object-oriented approach, and AspectML – a dynamic, functional approach to AOP. We illustrate the different features using the example of a logging aspect, adapted from [6].

The following code fragment shows how a static language like AspectJ implements a tracing aspect. The pointcut traceMethods selects the execution of any method of any class with any signature that is not itself executing within TraceAspect. The following advice retrieves the joinpoint signature to print to the log.

```
public aspect TraceAspect {

  pointcut traceMethods():
    execution(* *.*(..)) && !within(TraceAspect);

    before(): traceMethods()
    {
      Signature sig = thisJoinPointStaticPart.getSignature();

      System.out.println("Entering " +
          sig.getDeclaringTypeName()+" " + sig.getName());
} }
```

AspectS is a dynamic AOP implementation based on Smalltalk [3]. The code fragment below shows a method adviceLogging of an aspect object, which is called by the run-time weaver when installing the aspect. The method returns an AsBeforeAfterAdvice object that contains a set of AsAdviceQualifier objects, a set of AsJoinPointdescriptor objects to describe pointcuts and a beforeBlock that specifies the code to be woven. The run-time weaver uses this information to create wrapper methods for all specified joinpoint descriptors. This approach to run-time weaving precludes advising methods of system classes which are immutable at runtime, so that the pointcuts in the following example describe all methods of all subclasses of an InventorySystemRoot class.

```
adviceLogging
  | jpset classes |
  classes := InventorySystemRoot withAllSubclasses.
  jpset := OrderedCollection new.

  Classes
    do: [:each | each selectors
      do: [:eachSelector | jpset add:
      (AsJoinPointDescriptor targetClass: each
                            targetSelector: eachSelector ). ]].

  ^ AsBeforeAfterAdvice
      qualifier:
        (AsAdviceQualifier attributes: {#receiverClassSpecific})
      pointcut: jpset
      beforeBlock: [:receiver :arguments :aspect :client |
        Transcript show: ( receiver class ).]
```

In AspectML the example can be written in functional form. AspectML pointcut designators are untyped. Using the keyword "any" or specific function names, it is possible to advise the execution of functions, which must be identically typed.

```
advice before (| any |) (arg, s, info) = (print "Entering
"^(getFunName info)); arg )
```

Table 1 shows the main differences between AspectJ, AspectS and AspectML. The joinpoint model of AspectJ is much richer than that of either AspectS or AspectML. However, the latter two languages provide dynamic AOP capabilities, discussed in the following section.

Table 1. Comparison of selected AOP approaches

	AspectJ	**AspectS**	**AspectML**
Aspects can be instantiated	×	√	AspectML does not have an aspect construct.
Aspect inheritance	×	√	
Nested aspects	√	×	
Privileged aspects	√	×	
Polymorphic pointcuts	×	×	√
Polymorphic advices	×	×	√
Advice on field access	√	×	NA

3. DYNAMIC AOP

Dynamic aspect-oriented programming provides support for controlling aspects at runtime. This has some advantages:

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

The latter is a consequence of the typical implementation of dynamic AOP in which the core AOSD concepts are provided using the primary modularization concepts. The AspectS example above shows how advice and joinpoint descriptors are implemented as objects. Hence, they can be used to build generic libraries. Dynamic AOP is easier to implement in interpreted languages such as Smalltalk or ML, although a dynamic AOP versions of AspectJ exists [11]. A static language can approximate dynamic adaptation through run-time checks, such as in the following adaptation of the previous AspectJ example. Here, we have added a switch to turn the logging on and off.

```
public aspect TraceAspect {

  private static boolean loggingOn = false;

  public static void enable() {loggingOn=true;}
  public static void disable() {loggingOn=false;}

  pointcut traceMethods():
    execution(* *.*(..)) && !within(TraceAspect) && if(logingOn);

    before(): traceMethods()
    {…}
}
```

However, this is not a truly dynamic AOP system: For more complex control and configuration requirements, the complexity of conditional expressions increases rapidly; the control methods (enable and disable in the above example) must be called from the base system, which requires the base system to be aware of the aspects; and there remains an (however minimal) overhead of checking the configuration conditions. The following example shows how a dynamic AOP language such as AspectS allows dynamic control of aspect behavior:

```
AsAspect subclass: #AspectTraceD
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'AspectS-ShoppingCartDynamic'!

!AspectTraceD methodsFor: '…' stamp: '…'!

adviceLogging

^ AsBeforeAfterAdvice
   qualifier: (AsAdviceQualifier attributes:{#receiverClassSpecific})
   pointcut: [{
   AsJoinPointDescriptor   targetClass: AsInventoryD
                           targetSelector: #addItem:.
   AsJoinPointDescriptor   targetClass: AsInventoryD
                           targetSelector: #removeItem:.
   …. }]
   beforeBlock: [:receiver :arguments :aspect :client |
      … logging code here … ]! !
```

The base system and aspect extensions can be enabled and disabled separately, e.g. from a separate control thread, as shown in the following example:

```
|process1 test1|
process1 := [test1:=AsUserInterfaceD new test1 run.] newProcess.
process1 resume.

demoAspect:= AspectTraceD new.
demoAspect install.
demoAspect uninstall.

process1 terminate.
```

This also allows the reconfiguration of the aspect to adapt or configure the advice to changing requirements without losing state of the base system.

Dynamic AOP treats AOSD concepts as instances of the primary modularization concepts. For example, advice and pointcuts are objects in AspectS, and pointcuts are functions in AspectML, an extension of the functional language ML. An example taken from [4] is shown below. In this example, toLog of type pc(<a b> a~>b), is a pointcut that is passed as an argument to startLogger, an all purpose logging aspect.

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

Another example are pointcut objects in AspectS. In the following code, AspeptLogger is a generic logging subclass of AsAspect. It has a constructor method newJP that allows initialization with a set of AsJoinpointDescriptor objects. These are stored by the aspect and passed to the adviceLogging function that is called by the run-time weaver when installing the aspect. This allows us to build generic logging aspect that can be configured at runtime with the set of joinpoints to be logged.

Class Definition:

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

Class Method:

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

Instance Methods:

```
jpset: aJPDescriptor
   jpset := aJPDescriptor


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

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

Recent work on dynamic AOP has focused on solving a number of issues and problems that are not well suited for static AOP implementations. Handi-Wrap is a dynamic AOP extension for Java, which allows advice to be defined compositionally and supports run-time weaving [9]. PROSE *(PROgrammable extensionSions of sErvices)* is a dynamic AOP approach based on Java that allows aspects to be woven, unwoven, or replaced at run-time. PROSE supports rapid AOP prototyping and debugging and helps developers to understand the behavior of aspects in changed environment [10]. To address the call for recent demand for dynamic AOP, a new dynamic aspect weaver called Wool is presented in [8], which makes it possible to implement efficient dynamic AOP systems. Wool addresses the solution to the performance penalties caused in some prior implementations. An approach for language and platform independent dynamic AOP based upon reflection is presented in [11]. It focuses on dynamic adaptation of distributed systems at run-time. Dynamic AspectJ [12] considers the difficulties arising from the static scheduling strategy of AspectJ and shows how turning to a more dynamic strategy makes it possible to order, cancel, and deploy aspects at runtime.

4. ASPECT-ORIENTED MODELING

Aspect-oriented modeling (AOM) is an expansion of AOP to the upstream activity of software design. Most AOM techniques focus on providing modeling capabilities for the core AOSD concepts, usually as extensions to the Unified Modeling Language (UML). While there has been prior work on extending UML to AOM, most of the extensions expand UML either by introducing new meta-model classes or new notation elements without providing meta-level support.

Our approach as outlined here offers the following advantages:

- AOM is used within existing, mature software tools.
- Our model extension and any models produced by it can be exchanged between different MOF (Meta-Object-Facility) compliant UML modeling tools.
- All AOSD concepts are specified on the meta-level.
- Strict separation of base-model and cross-cutting concerns – the primary motivation behind AOP

AOM approaches can be distinguished along two orthogonal dimensions: the level of weaving and the symmetry of the approach. Our work is positioned at the asymmetric code-weaving level. The aspect-oriented model is converted to aspect-oriented code, which can be woven by an aspect-oriented compiler. We also make a clear distinction between the base-system and the cross-cutting concerns (Figure 1).

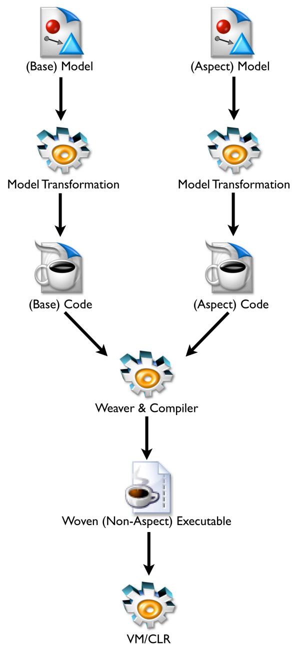

Figure 1 – Our AOP Approach in Context

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

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

However, this approach is not feasible for dynamic AOM, because dynamic approaches represent AOSD concepts as first-class modules. For example, joinpoint descriptors (pointcuts), advice and aspects are all objects in AspectS, while pointcuts are functions in AspectML. Thus, our approach will differ from the existing work in [5] by providing appropriate extensions to the Class meta-class for advice and pointcuts, as well as aspects (Figure 2 and Figure 3).

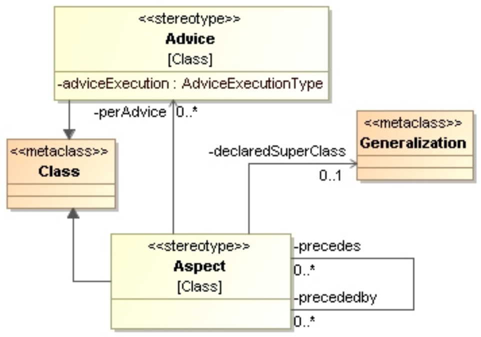

Figure 2 – Modeling Advice and Aspect

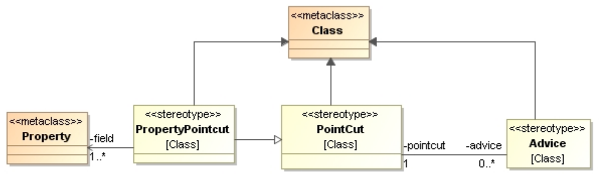

Figure 3 – Modeling Pointcut

Aspect instantiation, installation, de-installation, and configuration can then be modeled in the normal way using appropriately stereotyped objects.

We introduce the meta-class **CrossCuttingConcern** as a way of grouping related aspects. It extends the UML meta-class package, because cross-cutting concerns contain aspects in the same way as packages contain classes (Figure 4).

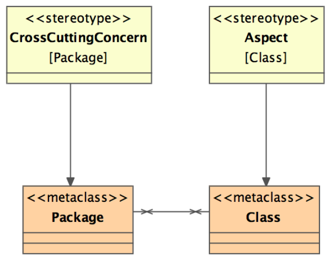

Figure 4 – Modeling Crosscutting Concern

Code generation can be done using standard MOF API, e.g. for Java based UML tools, or, alternatively, by working from the UML XMI (XML Model Interchange) format, the standard UML serialization. Both approaches use standardized mechanisms and are therefore compatible with existing modeling tools. Existing work in [5] has demonstrated the use of XSLT (XML Stylesheet Language Transforms) for generating XMI to AspectJ code. Our work-in-progress will leverage that mechanism. As a proof-of-concept, we implement an XSLT that generates valid code for our three target languages (AspectJ, AspectS and AspectML).

An overview of some of the prior work for modeling aspects in UML is presented in [13]. The early work is based on the extension mechanisms in UML 1.x versions. Because these mechanisms are not fully integrated with the meta-model, the specification of advices and pointcuts often remains in textual form [16],[17],[18], thus requiring special model parsers for code generation.

Initial work presented in [14] proposed the specification of aspects as stereotypes on classes and was later extended to include advice and pointcut specification [15]. It models cross-cutting associations to show which aspect features relate to which base-model elements, thus giving up a clear separation of aspects and base system, which is the primary objective of AOSD.

Other existing work is based on defining new UML meta-classes instead of defining stereotypes for existing meta-classes. This approach requires specialized tools to support the introduced meta-classes [19],[20].

5. CONCLUSION

In this poster presentation paper we provide an overview of diverse aspect-oriented concepts and their implementation in several programming languages. We compare the Joint Point Model present in these languages and discuss the benefits of dynamic aspects. Our approach to aspect-oriented modeling is platform independent. We provide code generation for AspectJ, AspectS and AspectML. The last two languages support dynamic aspects, whose modeling we support. The code generation currently relies on the modeler to verify the model. Although we present a number of OCL constraints as part of the model, others must be developed to support validation.

6. REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin (1997). Aspect-Oriented Programming, Proceedings of the European Conference on Object-Oriented Programming, vol.1241, pp.220–242.

[2] The AspectJ Team. AspectJ Programming Guide (v1.2). In http://aspectj.org

[3] R. Hirschfeld, Aspect-Oriented Programming with AspectS, in: Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7–10, 2002. Revised Papers, 2003.

[4] D. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A Polymorphic Aspect-oriented Functional Programming Language. ACM Transactions on Programming Languages and Systems. June 2008.

[5] J. Evermann. A Meta-Level Specification and Profile for AspectJ in UML. In Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, pages 27-49, August 2007.

[6] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Company, 2003.

[7] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-In-Time Aspect Weaver, Proceedings of the 2nd international conference on Generative programming and component engineering, vol. 48, 2003

[8] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming, Proceedings of the 1st international conference on Aspect-oriented software development, 2002

[9] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming, Proceedings of the 1st international conference on Aspect-oriented software development, 2002.

[10] N. Bencomo, G. Blair, G. Coulson, P. Grace, and A. Rashid. Reflection and Aspects meet again: Runtime Reflective Mechanisms for Dynamic Aspects, Proceedings of the 1st workshop on Aspect oriented middleware development,2005.

[11] A. Assaf, J. Noyé. Dynamic AspectJ, Proceedings of the 2008 symposium on Dynamic languages, Paphos, Cyprus, article no. 8, 2008.

[12] J. Davies , N. Huismans, R. Slaney, S. Whiting, M. Webster, and R. Berry. Aspect oriented profiler. In: 2nd International Conference on Aspect-Oriented Software Development. (2003)

[13] A. Reina, J. Torres, and M. Toro. Towards developing generic solutions with aspects. In: Proceedings of the AOM workshop at AOSD, 2004

[14] O. Aldawud, T. Elrad, and A. Bader. A UML profile for aspect oriented modeling. In: Proceedings of OOPSLA 2001, 2001

[15] O. Aldawud, T. Elrad, and A. Bader. UML profile for aspect-oriented software development. In: Proceedings of the AOM workshop at AOSD, 2003

[16] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. A UML notation for aspect-oriented software design. In: Proceedings of the AOM with UML workshop at AOSD, 2002

[17] M. Kande, J. Kienzle, and A. Strohmeier. From AOP to UML - a bottom-up approach. In: Proceedings of the AOM with UML workshop at AOSD, 2002

[18] M. Basch and A. Sanchez. Incorporating aspects into the UML. In: Proceedings of the AOM workshop at AOSD, 2003

[19] J. Grundy and R. Patel. Developing software components with the UML, Enterprise Java Beans and aspects. In: Proceedings of ASWEC 2001, Canberra, Australia, 2001

[20] H. Yan, G. Kniesel, and A. Cremers. A meta model and modeling notation for AspectJ. In :Proceedings of the AOM workshop at AOSD, 2004.