An Object-Oriented Framework for Aspect-Oriented Languages

Marko van Dooren [*] Eric Steegmans Wouter Joosen

IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium.

{marko,eric,wouter}@cs.kuleuven.be

Abstract

Aspect-orientation is a mechanism for modularizing cross-cutting concerns that has been added to many existing software engineering languages. The implementations of aspect-oriented language extensions, however, are typically tied to a specific base language. There is little or no code reuse between aspect-oriented extensions for different base languages, which makes these extensions difficult and expensive to build. In addition, existing software engineering tools do not work with the resulting aspect-oriented languages unless new plugins are developed.

We present Carpenter, an object-oriented framework for developing aspect-oriented language extensions. An aspect language is developed by reusing classes for generic language constructs from Carpenter, and writing subclasses of the abstractions in Carpenter to define new language constructs. An aspect weaver is created by implementing framework interfaces to weave language-specific constructs. The coordination of the weaving process is done by the Carpenter framework. Aspect languages developed with Carpenter get full IDE support with only a few lines of code. We have used our framework to create aspect weavers for Java, JLo, and AspectU.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks

General Terms Languages

Keywords Aspect, language, framework

1. Introduction

Aspect-orientation is an increasingly used technique to modularize cross-cutting concerns. Originally developed for programming languages, with AspectJ [20] as the main implementation, the technique is now being used in more and more languages in various stages of the software development process. Aspect-orientation has been added to use cases [17, 19, 33], to architectural description languages [10, 25, 27, 28], and to many programming languages such as C++ [34] and C# [32]. In addition to adding aspect-orientation to existing languages, there is also a need to add new aspect-oriented capabilities to existing aspect-oriented languages [13, 22, 24, 36].

The implementations of the aspect-oriented extensions of most languages, however, are typically written from scratch. This is an error-prone and costly way to create an aspect language. To make matters worse, advanced programming tools, which are essential in a modern software development process, do not work with the aspect-oriented versions of the languages they were designed for. Plugins must be developed to provide tool-support for the new aspect language.

Fradet and Südholt [8] already recognized in 1998 that aspect weaving can be performed using a general transformation framework because the mechanism is always the same. A cross-cutting concern is captured as an advice that is woven into the model at the places (join point shadows) specified by pointcut expressions.

A number of approaches exist to improve the development of aspect-oriented languages. SourceWeave.NET [18], Weave.NET [21], LOOM.NET [32], and Aspect.NET [30] exploit the common language infrastructure of the .NET platform to add aspect-orientation to a language. These approaches, however, are inherently limited to programming languages supporting .NET. In addition, they force programmers to write aspects using language constructs of the common intermediate language (CIL), which is not appropriate for logic or functional programming. The abc compiler [1] and Reflex AOP [35] simplify the development of aspect languages, but the host language is limited to Java, or an extension of Java. Roychoudhury et al. [29] present a model-driven approach for creating aspect languages. Their approach is generic, but the developer of an aspect language must work with multiple transformation languages, and transformations that generate other transformations.

The contribution of this paper is the development of an object-oriented framework for developing aspect-oriented languages, called Carpenter. In our approach, aspect lan-

[*] Post-doctoral researcher of the Fund for Scientific Research - Flanders.

guages are developed by implementing only the language-specific constructs and their accompanying weavers directly using standard object-oriented techniques. Common language constructs for aspect languages, and most of the infrastructure for aspect weavers are provided by Carpenter. To provide more specialized support for families of languages, paradigm-specific layers can be developed. The current Carpenter framework contains such a layer for object-oriented programming languages. We evaluated the Carpenter framework to develop aspect-oriented extensions of Java, JLo, and a language for use cases.

Outline

The remainder of the paper is structured as follows. In Section 2 we discuss the requirements for easily creating aspect-oriented languages. In Section 3 we give an overview of the Carpenter framework, which is discussed in more detail in Section 4. We discuss the creation of parsers in Section 5. We evaluate our approach in Section 6 by implementing a aspect-oriented extensions for Java, JLo, and a language for use cases. We discuss related work in Section 7, and conclude in Section 8.

2. Requirements

In this section, we discuss the requirements for simplifying the development of aspect-oriented languages.

In the process of developing aspect-oriented languages, we identify three main stakeholders: the developer of the base language, the developer of the aspect-oriented extension of a language, and the developer of an aspect weaver for an aspect-oriented language. Note that the second and third stakeholders are not necessarily the same. For example, multiple aspect weavers can be created for a single aspect-oriented language to perform different kinds of optimizations, such as maximizing the execution speed of the woven program, or minimizing its size.

To simplify the development of aspect-oriented languages, we identify the following main requirements for the different stakeholders. The involved stakeholders are shown between parentheses for each requirement.

1. **A language- and paradigm-independent approach:** *(all)* The approach must be applicable to all types of languages, from requirements engineering languages to domain specific programming languages.
2. **Modularity of aspect weavers:** *(aspect weaver developer)* Similarly, the aspect weaving mechanisms for many aspect-oriented languages have much functionality in common. Such common functionality should be provided in reusable modules that can be composed by the developer of an aspect-oriented language. For example, the high-level orchestration is always the same: find join point shadows, sort the advices per shadow, and perform the actual weaving. The developer for a particular aspect-oriented language extension should not have to implement this process from scratch.
3. **Modularity of aspect-oriented language constructs:** *(aspect weaver developer, aspect language developer)* Multiple aspect-oriented languages can have language constructs in common, such as generic pointcut expressions and advice types. The developer of an aspect-oriented language extension must be able to reuse the implementations of the semantics of these language constructs in a modular way.
4. **Modularity of the base language:** *(all)* The semantics and the parser for the base language can be very complex. Therefore, their implementations should be reusable to build language extensions. The semantics of language extensions should be plugged into the semantics of the base language. In addition, if a base language is changed (for example in a new version) in a way that does not affect an extension of that language, then that extension should not require any changes to work with the new version of the base language.

Requirements 2 and 3 apply at multiple levels in the language hierarchy. For example, aspect-oriented extensions of languages that belong to a particular family, such as object-oriented programming languages, should be able to share common pointcut expressions and weaving functionality. Similarly, an aspect-oriented extension of JLo, which itself is an extension of Java, should be able to reuse the implementation of the aspect-oriented extension of Java.

For requirements 2, 3, and 4 the set of stakeholders becomes bigger with every step. This reflects the stack structure of the process. An aspect weaver is built for an aspect language, which is built for a base language. Reusing aspect weaving functionality across different aspect-oriented languages is very difficult if those languages cannot share language constructs. Reusing aspect-oriented language constructs and their semantics is very difficult if the base languages cannot share language constructs.

3. An Object-Oriented Approach

Our approach is based on the Chameleon framework, which is a generic object-oriented framework for language development. Figure 1 illustrates the architecture of our approach. Layers with a thick black border are part of Carpenter, while layers with a thick gray border are part of Chameleon. To keep the figure simple, arrows for *uses* relations are only drawn towards the most specific layer that is used by another layer. The *super* layers are used as well.

The top of the language hierarchy is the Chameleon layer for generic language constructs. Language constructs do not only contain the structure of a program – like AST nodes – but they also encapsulate the static semantics. For example, each language construct has a `verify` method to determine whether it is valid, such tool developers do not have

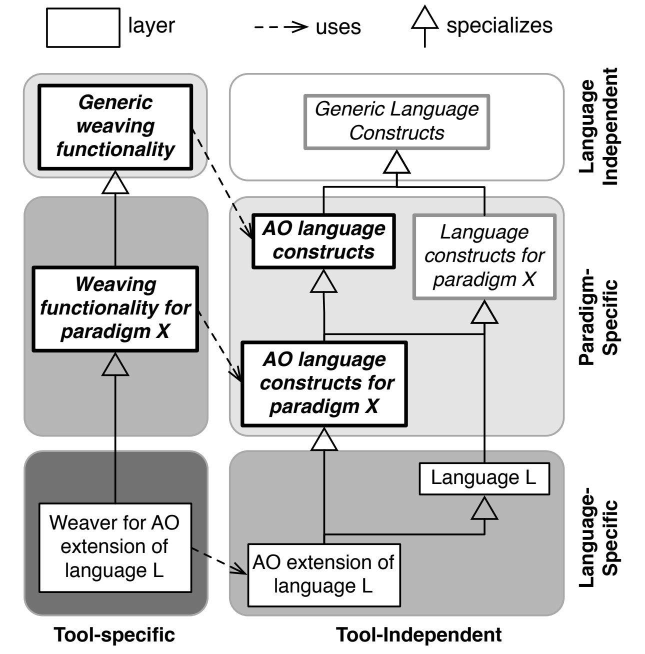
layer
uses
specializes
Generic weaving functionality
Generic Language Constructs
Language Independent
AO language constructs
Language constructs for paradigm X
Weaving functionality for paradigm X
AO language constructs for paradigm X
Paradigm-Specific
Language L
Weaver for AO extension of language L
AO extension of language L
Language-Specific
Tool-specific
Tool-Independent

Figure 1. The architecture of Carpenter.

to duplicate these semantics. The top layer also contains many abstractions for similar language constructs. For example, a `Declaration` is any language construct that has a `Signature`, such as a type, a method, or a use case. These abstractions greatly improve the language-independence of software engineering tools.

Below the top layer, there are paradigm-specific layers. Top layer of the Carpenter framework provides classes for generic aspect-oriented language constructs. Many pointcut expressions can implemented directly in this layer by using the abstractions provided by the top layer. The `within` pointcut expression, for example, matches any element that is lexically within a certain `Declaration`. As such, this pointcut expression can be reused directly in any aspect-oriented language. The top layer of Carpenter also provides abstractions to improve the language-independence of aspect-oriented tools.

Multi-paradigm layers provide additional language constructs for language that belong to a specific combination of paradigms. For example, we have implemented a layer in Carpenter for aspect-oriented extensions of object-oriented programming languages. This layer contains for example a pointcut expression for method invocations.

At the bottom of the hierarchy are the implementations of concrete languages. A concrete aspect-oriented language `AO-L` reuses both the implementation of the base language `L`, and the implementation of the aspect-oriented language constructs for the paradigm of `L`. Aspect-oriented language constructs that are specific for `L` are implemented in `AO-L`.

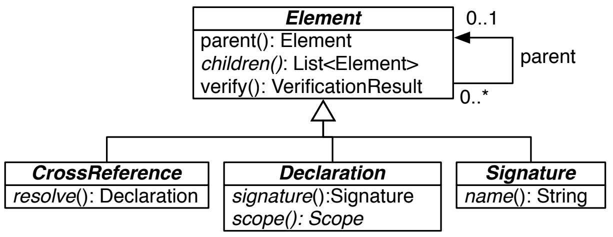

Figure 2. A few core abstractions of Chameleon.

The hierarchy for aspect weavers – or any other tool – follows the same layering structure. The language-independent weaving code is written in the top layer, while subsequent layers add support for weaving specific paradigms of language or specific languages. For example, the top layer of the aspect weaver of Carpenter contains the code to coordinate the weaving process. It uses abstractions such as `PointcutExpression` to find join point shadows without having to know which aspect-oriented language is used. The actual weaving is performed by lower layers since that requires paradigm-specific or language-specific knowledge.

3.1 Core Abstractions of Chameleon

Figure 2 shows a simplified class diagram of the most important classes of Chameleon that are used in Carpenter. Every language construct implements the top interface `Element`, which has methods for navigating the lexical structure of a model. The `verify` method checks whether or not an element is valid.

A `Declaration` is any element that has a signature (name). Examples of declarations are methods, class, aspects, use cases, and so forth. A `Signature` has at least a string that represents its name, but can also contain additional information such as parameter names and types.

A cross-reference is any element that references a declaration. Examples are method invocations, type names, and so forth. The `resolve` method encodes part of the semantics of the cross-reference by computing which declaration is referenced. The Chameleon IDE, for example, uses this method to support navigable hyperlinks without having to know which concrete language is used.

4. A Framework for Aspect Languages

In this section, we discuss the most important classes of the Carpenter framework. In section 4.1, we discuss the top layer for aspect-oriented languages. In section 4.2 we discuss the top layer for aspect weavers. In section 4.3 we discuss the layer for object-oriented languages.

Because of space concerns, the class diagrams that we use throughout the paper are *simplifications* of the real framework. As such, we omit all elements that are not required for explaining our approach.

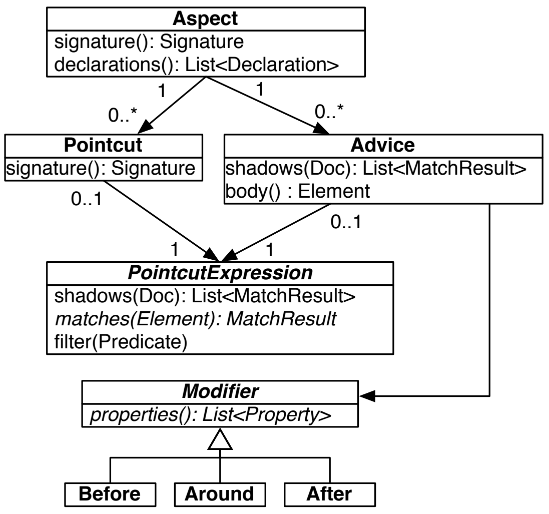

Figure 3. Aspects.

4.1 Abstractions For an Aspect-oriented Language

Figure 3 shows the generic classes to model aspects. An aspect has a name, pointcuts, and advices. A pointcut has a name and a pointcut expression. An advice has a body, which in general can be any kind of element, and a pointcut expression. The types of advice are modeled as modifiers and accompanying properties, which are omitted to save space. Neither a pointcut nor an advice can have formal parameters in the top layer of the framework, since they are not available in all languages. Support for parameters is added in the layer for object-oriented languages.

Figure 4 shows a number of language constructs of the generic pointcut language. The semantics of a pointcut expression is defined by its `matches` method, which determines whether the pointcut expression matches a potential join point shadow. Note that redefinitions of the `matches` methods are not shown to save space. The `filter` method is used to compute the pointcut residue, which is the dynamic subtree of a pointcut expression.

The generic pointcut language supports disjunction, conjunction, and negation, and provides classes for a number of concrete pointcuts. A pointcut expression (`PointcutRef`) delegates join point shadow matching to an existing pointcut. It matches each element that is matched by the referenced pointcut. The `CrossRefPointcutExpr` class selects join point shadows that are cross-references and that reference a declaration that satisfies a certain pattern. Similarly, the `Within` expression matches elements that are lexically defined within a declaration that satisfies a pattern. A number of typical patterns for aspect-oriented programming are provided by the framework. The regular expression and wildcard patterns constrain the signature of a declaration, while the container pattern puts a constraint on the nearest lexical parent declaration of a certain type. This can be used for example, to match references to a field name `f` of class `T`.

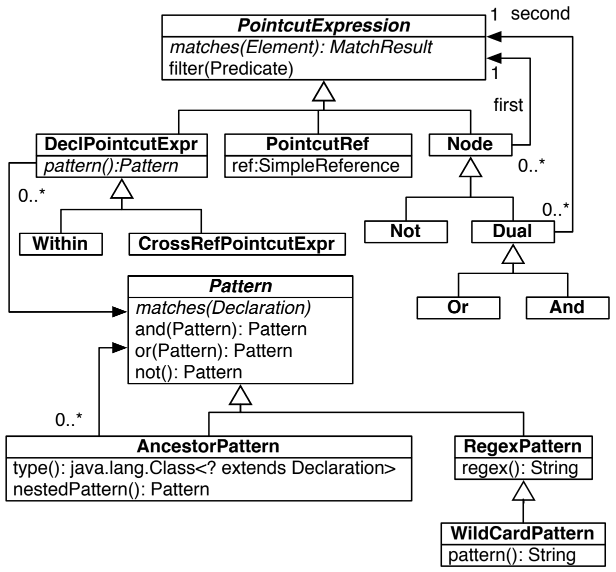

Figure 4. Pointcut expressions.

It is important to note that the leaf classes in Figure 4 are concrete and have only dependencies with classes from the top-level Chameleon layer. Therefore, the classes shown in Figure 4 can be used regardless of the language to which aspect-orientation is added. The developer of the aspect language must only add classes for new language constructs.

4.2 The Weaving Process

In this section, we present the generic weaving infrastructure of Carpenter. It is important to note, however, that whereas the generic aspect language of the previous section resembles a library due to the amount of concrete classes, the weaving infrastructure is more like a real framework with many abstract classes. This is because selection of join point shadows can be done easily based on high-level abstractions, but the actual weaving requires construction and manipulation of elements at the level of the concrete language.

Figure 5 shows the top classes of the aspect weaver infrastructure. The `AspectWeaver` class represents the entire aspect weaver. The actual weaving is delegated to a linked list of `Weaver` objects that implement the *Chain of Responsibility* design pattern [9]. To define the order in which multiple aspects are applied to a single join point shadow, the aspect weaver uses a sorting strategy. Each aspect language defines a concrete subclass of `AspectWeaver` that initializes the weaver chain and the sorting *Strategy*. Each subclass of `Weaver` is responsible for weaving one or more combinations of a type of advice and and type of join point shadow.

The weavers do not immediately perform the weaving because the order in which the aspects for a particular join point shadow must be woven does not correspond to the order in which advice objects are selected by the as-

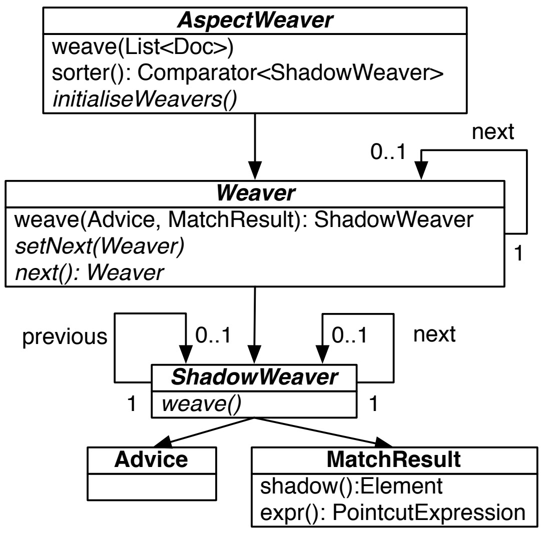

Figure 5. The aspect weaver infrastructure.

pect weaver. Instead, they create `ShadowWeaver` objects. A `ShadowWeaver` is responsible for weaving one particular advice for a particular join point shadow. A `ShadowWeaver` contains a reference to the advice that must be woven and a `MatchResult` object. A `MatchResult` object is created during selection of the join point shadows and keeps a reference to the shadow and the pointcut (sub)expression that matched the shadow. The reference to the pointcut (sub)expression is stored to allow support for dynamic pointcut expressions. For such pointcut expressions, it is necessary to know exactly which part of the pointcut expression of an advice matched the join point shadow such that the correct dynamic code can be inserted. For example, if pointcut expression `within(T) & if(f()) | within(S) & if(g())` matches within `T`, only code for evaluating `f()` should be inserted. The `ShadowWeaver` objects for a particular shadow will be connected to form a doubly linked list. This allows the various advices for a single join point shadow to be chained together correctly during the actual weaving.

Figures 6, 7, and 8 illustrate the weaving process. The solid arrows in collaboration diagrams denote references that are stored in fields, whereas the striped arrows denote references via local variables. Note that the messages in the collaboration diagrams are not exact representations of the code in the classes of Figure 5 since these classes are abstract. For example, the `create` call will be performed by a subclass of `Weaver`, and will create an object of a subclass of `ShadowWeaver`.

For each advice in the model, the aspect weaver asks the advice to which join point shadows it must be applied. The advice delegates this call to its pointcut expression, which encapsulates the semantics of join point shadow matching. The resulting `MatchResult` objects (`mr` in the figure) are then given to the chain of weavers, together with the advice.

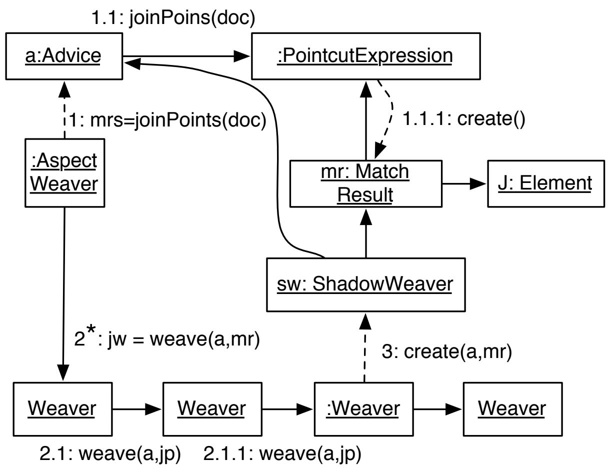

Figure 6. Phase 1: creating shadow weavers.

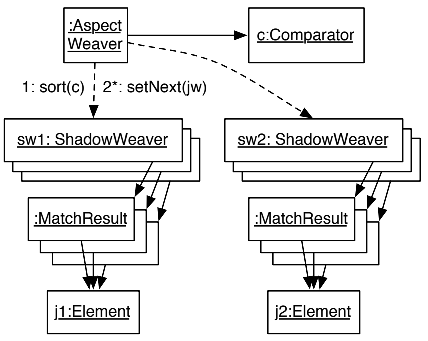

Figure 7. Phase 2: sorting the shadow weavers.

In this figure, only the call for weaving a single combination of advice and match result is shown. In this case, the third weaver decides it is responsible for weaving this combination, and creates a shadow weaver that will perform the actual weaving.

After passing all join point shadows to the chain of weavers, the aspect weaver builds a map with the shadows as keys, and a collection of `ShadowWeaver` objects as the value. These collections are then sorted, and the shadow weavers are linked together.

Finally, the `weave` method is invoked on the first shadow weaver to start the actual weaving process. Each shadow weaver first passes control to the next one in the list, and then weaves its own advice. The actual weaving is performed in two steps. In the first step, any infrastructure is generated to store the advice. This can for example be a method. In the second step, the join point shadow is transformed to incorporate the advice, for example an expression can be replaced with an invocation of the method that contains the advice code. The second step is only performed by the first shadow weaver in the chain. Every shadow weaver `N` beyond the first one will instead return an element that allows the previous shadow weaver `P` in the chain to incorporate the

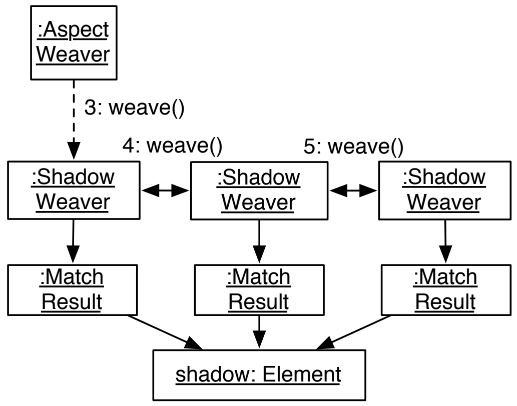

Figure 8. Phase 3: performing the weaving.

advice woven by N in the advice woven by P. Therefore, all shadow weavers restrict the kind of shadow weaver that they can be connected to, such that they know how to process the result of the next shadow weaver. We do not consider this to be a severe restriction since we do not know of a situation where this would be inappropriate

4.3 The Object-Oriented Layer of Carpenter

The OO layer of Carpenter provides additional support for adding aspects to object-oriented programming languages and for the corresponding weavers. The code in this layer assumes that the host language is an object-oriented programming language, and can therefore rely on abstractions defined in the OO layer of Chameleon.

4.3.1 A Pointcut Language for OO Languages

Figure 9 shows the support for exposing context information via parameters is provided. Both pointcuts and advice get a list of formal parameters, which are defined in the Chameleon OO layer. The pointcut expressions that expose parameters are taken from the AspectJ pointcut language. They are used within the shadow weavers to insert local variable in the generated code. In addition, a pointcut expression is added to reference a pointcut that has parameters.

Support for matching method and constructor invocations, and field reads is provided through new subclasses of `DeclarationPattern` that are similar to those of Figure 4. For reasons of space, we do not show a separate class diagram for the new patterns. These patterns can be used together with the pointcut expression for cross-references, which is defined in the top layer of Carpenter.

4.3.2 Advice Weaving for OO Languages

Support for advice weaving is limited to providing helper classes to perform the weaving. Generating the actual language specific code is the responsibility of the language module of the specific aspect language. The OO layer of Carpenter offers classes to facilitate weaving advice for expressions and statements. TO create language constructs that

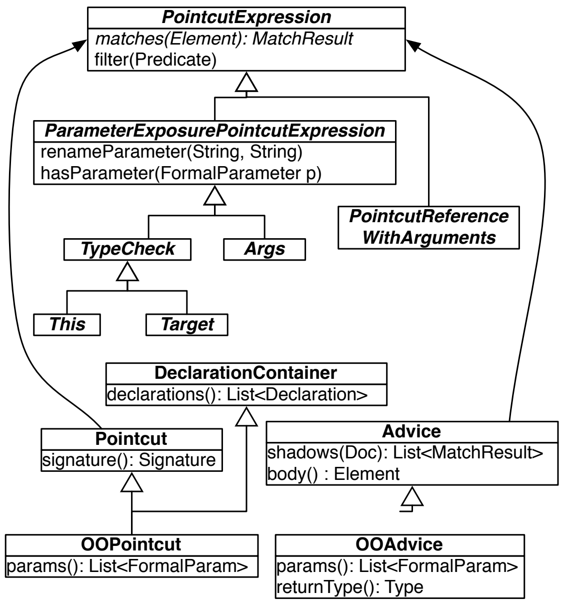
PointcutExpression
matches(Element): MatchResult
filter(Predicate)
ParameterExposurePointcutExpression
renameParameter(String, String)
hasParameter(FormalParameter p)
PointcutReference WithArguments
TypeCheck
Args
This
Target
DeclarationContainer
declarations(): List<Declaration>
Pointcut
signature(): Signature
Advice
shadows(Doc): List<MatchResult>
body() : Element
OOPointcut
params(): List<FormalParam>
OOAdvice
params(): List<FormalParam>
returnType(): Type

Figure 9. Exposing context in pointcuts and advice.

are represented differently in different object-oriented languages, such as `if-then-else` statements and exception handlers, factories are used.

The OO layer also contains classes for orchestrating the binding of context information in the form of parameters. Shadow weavers for pointcut expressions that support parameter exposure the `RuntimeContextProvider` interface. This interface is used by the parameter binder class of the OO layer to add parameters to the advice and bind them to the appropriate values in the context.

5. Creating Parsers for Aspect Languages

Gray et al. identify parser construction as one of the main challenges for constructing aspect weavers [11]. First, there is a need for a good parser for the base language. Second, the parser for the base language must be extended with a syntax for defining aspects.

In this section, we report on our experience with using the ANTLR [26] parser generator to construct parser for aspect-oriented languages. ANTLR supports composition of grammars, and grammar files are available for many existing languages. Other parsing technologies can be used, though, since Chameleon hides parsing behind a generic interface.

Figure 10 illustrates how a parser for an aspect-oriented extension of Java is created in ANTLR. In this figure, the double arrows denotes imports. The grammar of Java is defined in two files: one for the lexer (JavaL.g) and one for the parser (JavaP.g). The parser file contains the syntax rules for Java, along with semantics actions that create on object representation of the model. To construct an actual Java parser, both files are imported in a root parser file (Java.g). To add syntax definitions for aspect-oriented programming, both the

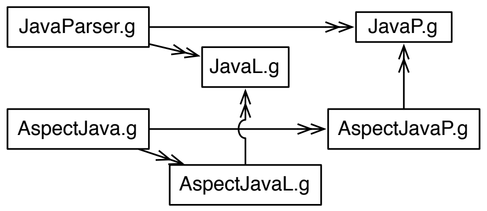

Figure 10. Extending a parser to support aspects.

lexer and parser are extended and the recombined into a new parser. Extending the lexer and parser of Java is done by importing them in the lexer and parser of AspectJava, and then overriding and adding syntax definitions.

Multiple inheritance in ANTLR should make it possible to reuse the grammar rules for generic aspect-oriented language constructs, but a bug in ANTLR prevented us from writing the required grammar compositions. In addition, ANTLR does not allow the addition of a case to a syntactic rule in a modular way. This is needed for example to add `proceed` to the existing Java expressions such that it can be used in advice code. Currently, either all existing cases of the rule for expressions must be duplicated in the overriding definition, or the original Java must be refactored. We plan to experiment with PPG [3] or Rats! [12] to reuse the grammar rules for generic aspect-oriented language constructs.

6. Evaluation

In this section we evaluate Carpenter by building a number of aspect oriented language extensions and measuring the amount of work that is required. We first discuss the evaluation approach in Section 6.1. We then discuss the three aspect-oriented languages that we developed in Sections 6.2, 6.3 and 6.4. Finally we summarize the results of the case studies by revisiting the requirements in Section 6.5.

6.1 Evaluation Approach

We have used the Carpenter framework by building aspect weavers for the Java and JLo, which is an extension of Java. We have also developed a weaver for the AspectU language, which adds aspect-orientation to use cases. The implementations of these languages and the Carpenter framework are available online [37].

We use the size of the code base for each extension as an indication of the amount of work that is required to create the extension. Therefore, the line counts do not include comments, generated code (such as imports), and lines that contain only braces or brackets. Figure 11 shows the size of the generic and object-oriented layers of Chameleon and Carpenter, along with the size of the Chameleon IDE. The bold rows in the tables in this section are used for code that is related to aspect-orientation.

The Chameleon Eclipse IDE is an Eclipse plugin that uses Chameleon for modeling the source of a project. The IDE uses only a few abstractions of Chameleon to support a number of essential features for modern IDEs. An outline shows a tree structure of the declarations in a file. Navigable hyperlinks allows a user to click on a cross-reference after which the IDE jumps to the definition of the referenced declaration. Errors in the model are reported by underlining the problem region in red and adding an entry to the *problem view* of Eclipse. Language-specific requirements of the IDE are hidden behind interfaces that must be implemented to provide support for a concrete language. This mostly concerns parsing, providing meta-information such as the positions of elements, and optionally custom visualizations.

6.2 An Aspect-oriented Extension of Java

We have implemented an aspect weaver for Java 1.5, which we call AspectJava in the remainder of this paper. The AspectJava weaver does not generate bytecode, but generates Java source code instead. The supported advice types are: *after*, *before*, *around*, *after returning*, and *after throwing*. AspectJava supports the following pointcut expressions: calls of methods based on their signature or annotation, field reads, catch clauses, delegation to pointcuts, elements within a certain type or method, class cast expressions, dynamic pointcut conditions (`if`), run-time condition based on the type of arguments, and the target or receiver of a message.

The advice infrastructure factories for all expressions generate static methods that contain the advice. Each shadow weaver provides support for weaving multiple advices into a single join point shadow by generating an invocation of the static method generated by the next shadow weaver in the chain. Passing of arguments and invoking the original method call (if the join point shadow is a method or constructor invocation) is done using reflection.

The motivation for using reflection was to make it easier to get the generated Java source code accepted by the Java type checker. The downside of using reflection in the generated advice infrastructure, however, is that the code of the shadow weavers becomes less reusable. The reflective capabilities of for example Java, Smalltalk, and C++ are too different to be able to extract much common code. The static structures of these languages, however, have much more in common. Implementing a factory to generate for example, a method that behaves like a static method in Java is much

	LOC
Top layer of Chameleon	4644
Chameleon Eclipse IDE	6989
Top layer for aspect-oriented Languages	**856**
Top layer for aspect weavers	**433**
OO layer of Chameleon	7640
OO layer for aspect-oriented languages	**895**
OO layer for aspect weavers	**234**

Figure 11. Line counts for Chameleon and Carpenter.

Java language	5183
Java grammar file	1170
Java plugin for Chameleon IDE	223
AspectJava language	**126**
AspectJava weaver	**1438**
AspectJava grammar file	**151**
AspectJava plugin for Chameleon IDE	**39**

Figure 12. Line counts for Java and AspectJava.

easier. In addition, such factories are also reusable for creating other tools such as a refactoring tool. Having a refactoring tool that generates reflective code, even if it does so correctly, does not seem like a good idea. The use of reflective code also prevented us from implementing the functionality for exposing context information in Carpenter. The orchestration of the process is done in Carpenter, but since formal method parameters are part of the context information, the code mechanism for context information also suffers from our choice to generate reflective code. An additional problem is that the code for generating the reflective Java code in some of the shadow weavers for run-time pointcut expressions is hard to read.

The table in Figure 12 shows the line counts for AspectJava and the language module for Java. It is clear from these numbers that the framework approach works very well for the definition of the AspectJava language itself. Most aspect-oriented language constructs can be reused from CarpenterThe entire base language is reused from the Java language module. The code for Java specific aspect-oriented language constructs (126 LOC) is less than 7% of the total code for the aspect-oriented language constructs in AspectJava. Since AspectJava reuses virtually all code for aspect-oriented language constructs from the generic and OO layers of Carpenter, this total is 126+895+856=1877 LOC. In reality this number is even better since it does not yet include code that is reused from Chameleon, such as code for resolving cross-references. The precise amount of code that is reused from Chameleon, however, is hard to count.

For the aspect weaver, the percentage of reused code is significantly lower. Only about a third of the code for the AspectJ weaver could be reused – it uses virtually all code from the Carpenter weaving layers. While this is still a good result, we expect that reuse can be improved further by generating regular code instead of reflective code. We expect that generating regular code will allow more generic OO code to be moved to the OO layer of Carpenter.

The line count parser for the ANTLR grammar files is calculated in a similar way. Grammar reuse is very good, only two rules from the Java grammar had to be overridden: `compilationUnit` to add aspects, and `expression` to add `proceed`. As such, there is still some duplication, but it only concerns about 20 lines of code.

	LOC
JLo language	1663
JLo to Java compiler	1703
JLo grammar file	233
JLo plugin for Chameleon IDE	92
AspectJLo language	**9**
AspectJLo weaver	**267**
AspectJLo grammar file	**235 (86)**
AspectJLo plugin for Chameleon IDE	**37**

Figure 13. Line counts for JLo and AspectJLo.

A remarkable result is that IDE support is virtually free. Plugin functionality that is specific for Java – mostly code for visualizing method signatures – is reused by extending the plugin for Java. The two classes in the AspectJava plugin take only 39 lines of code, 14 of which are methods for creating user interface strings such as the version number. The outline shows the aspects and pointcuts in a file, using a reference to a parameter of a pointcut or advice as a hyperlink make the cursor jump to its definition, and syntactic and semantic errors are reported. Not a single line of code was written for AspectJava to support these features.

It is hard to compare line counts of our approach with those of other approaches, but Avgustinov et al. report 167 lines of code to add support for matching and weaving cast join point to AspectJ with the abc compiler [1]. We need 176 lines for the cast join point, so the effort is similar. The advantage of abc is that is offers advanced optimization of the woven code for Java. The advantage of our approach is that it is much more generic.

6.3 An Aspect-Oriented Extension of JLo

To test the extensibility of AspectJava, we developed an aspect weaver for JLo, which adds a dedicated composition relation to Java [38]. It is not in the scope of this paper to discuss the benefits or full semantics of this composition relation. What is important is that JLo extends Java with new language constructs that have a significant influence on the lookup mechanism of the language. AspectJLo is implemented as a layer that specializes AspectJava, and can therefore reuse all functionality defined for AspectJava.

The aspect weaver for AspectJLo supports all of the advice types and pointcut expressions of AspectJava, but adds a pointcut expression for subobject reads. These are similar to field reads, but the AspectJava weaver cannot process them since it has no knowledge of subobjects.

The table in Figure 13 shows the size of the AspectJLo implementation. The sizes of the JLo components are included for reference. The AspectJLo language module contains only a subclass of `Language` to represent the AspectJLo language. No new language construct is needed for subobject reads because they can be modeled directly with the Carpenter classes of Figure 4.

	LOC
Use case language	2665
Use case grammar file	435
Use case plugin for Chameleon IDE	33
AspectU language	**72**
AspectU weaver	**125**
AspectU grammar file	**104 (64)**
AspectU plugin for Chameleon IDE	**39**

Figure 14. Line counts for use cases and AspectU.

For AspectJLo, the Carpenter framework allowed us to define an aspect weaver with very little work. Because JLo is a Java extension, the use of reflection in the generated code caused no problems, and we could reuse the complete AspectJava weaver. Only the code for matching and weaving subobject reads must be written. Everything else is reused from the AspectJava weaver.

The grammar definitions in AspectJava could not be reused because we could not get the multiple inheritance mechanism of ANTLR to work correctly, as mentioned in Section 5. Therefore, the AspectJLo grammar extends the JLo grammar, and the syntax definitions for aspects and pointcuts are copied from the AspectJava grammar. Otherwise, the grammar would only be 86 lines long.

As with AspectJava, obtaining support for the Chameleon IDE requires some trivial configuration code.

6.4 An Aspect-Oriented Extension of Use Cases

To study how well Carpenter works for a non-programming language, we implemented AspectU [33]. AspectU is an aspect-orientated extension of a language for use cases, and provides support for pointcut expressions for matching steps, use cases, and use case extensions. All use cases, steps, and extensions in the base language are annotated with a name that can be used in the pointcuts. Context information is exposed via the pointcut expression `bind(var,val)`. An advice consists of a list of steps and a list of extensions that can be added to a use case. A special `proceed` step can be used in around advice.

The table in Figure 14 shows the size of the AspectU implementation. The base use case language is included for reference. Note that our base use case language is more advanced than that of AspectU. Both the implementations of the AspectU language and the AspectU weaver are very small. The fact that the semantics of the `bind` pointcut expression of AspectU can be expressed as simple text substitution makes it much easier to insert the context information than is the case for Java. As with AspectJLo, the grammar definition duplicates grammar rules for generic aspect-oriented language constructs. The line count between parenthesis shows the size of the grammar if we could reuse those definitions. Again, IDE support is virtually free.

6.5 Conclusion of the Case Studies

In this section, we revisit the requirements that we presented in Section 2, and summarize the results of the case studies.

1. **A language- and paradigm-independent approach:** By developing aspect-oriented extensions of two programming languages and a language for use cases, we have shown that the approach works for languages in two completely different paradigms. We found no indications that the approach would not work for other paradigms.
2. **Modularity of aspect weavers:** The framework approach worked well for the aspect weavers. All aspect weavers reuse the framework classes of Carpenter. In addition, the AspectJava weaver reuses functionality from the OO layer of Carpenter, and the AspectJLo weaver reuses the complete AspectJava weaver.

   The developed languages also revealed a number of limitations of our approach. First, the generation of reflective code in the AspectJava weaver prevents reusing that code for other object-oriented languages because the reflection mechanisms of these languages differ too much. An important research challenge is to study whether generating regular code can lead to reusable weaving code for object-oriented languages.

   Second, the current weavers insert the advice body in some form for every matched join point shadow, which is problematic for large programs. Such optimizations can be implemented in the language-specific weavers, but an important research challenge is to study how they can be supported by Carpenter.
3. **Modularity of aspect-oriented language constructs:** The framework approach worked very well for defining the aspect languages. Only a few language-specific elements had to be implemented for each language. The rest of the language could be used from Carpenter. AspectJLo completely reuses the language definition of AspectJava.

   ANTLR has proven helpful for creating parser for the aspect languages, but did not result in fully modular parsers. The inability to add cases to an extended grammar resulted in some duplicated cases. In addition, problems with multiple inheritance prevented the extraction of the common grammar rules for the generic aspect-oriented language constructs.
4. **Modularity of the base language:** Aspect-oriented language constructs were be added to the base language without modifying the latter. The base language semantics are completely reused. Even though the concrete aspect-oriented languages contain no code for name resolution, the lookup mechanisms of the base language still works within an advice body still works. In addition, variable names in the base language code resolve to a parameter of an advice block if the name matches, even though the base language has no knowledge of advice.

Carpenter significantly reduced to work to define aspect-oriented extensions of Java, JLo, and our use case language. Most of the aspect-oriented language constructs and the code to orchestrate the weaving process could be reused from Carpenter. For the aspect weavers, a significant amount of code could be reused but we think that the support for weavers can be further improved. The use of Chameleon in the aspect-oriented languages made it very easy to obtain IDE support. Only a few lines of code were needed for each language, giving support for syntax highlighting, an outline, navigable hyperlinks, and error-reporting.

The development of Carpenter also revealed a few shortcomings in Chameleon. Carpenter revealed the need for a generic mechanism for modifying `Declarations`. This is needed to support for example inter-type declarations. Such a mechanism is implemented for particular elements, but is not yet available in general. In addition, the builder infrastructure had to be modified because the aspect weavers must know which parts of a model represent the user project in which aspects must be woven, and which parts (if any) represent unmodifiable elements such as the language library.

7. Related Work

Roychoudhury et al. present a model driven approach for construction of aspect weaver[29]. They identify four main challenges: 1) parser construction , 2) weaver construction , 3) accidental complexity of transformations, and 4) language-independent generalization of transformations. The authors address challenges 1 and 2 by using program transformation techniques. They address challenge 3 by defining an abstract layer for aspect-orientation (called GAspect), and using ATL transformations to generate RSL program transformation rules which incorporate low-level language details. This allows the aspect developer to focus on the language concepts without dealing with low-level details. Challenge 4 is addressed by using higher-order transformations, which allow reuse across multiple aspect languages. The authors construct aspect weavers for FORTRAN and Object Pascal. In our approach, challenge 1 is addressed by using a parser generator, challenge 2 by using Carpenter, and challenges 3 and 4 by using Chameleon. The main differences with our approach are the following. First, the metamodels in Carpenter encapsulate the language semantics instead of having them spread over data models, transformations, and program analyzers. Second, we implement the aspect languages and weavers directly using object-orientation instead of defining transformations to generate other transformation rules.

JastAdd [7] provides a DSL and accompanying tools for implementing languages. The AST structure is defined in an attribute grammar from which corresponding AST classes are generated. Additional functionality is implemented in inter-type declarations and woven into the AST classes. Similar to Chameleon, a language can reuse elements from other languages. But JastAdd does not provide a library of generic abstractions. As such, the implementation for Java and a use case language would share nothing. But even if such abstractions were defined, it would be impossible to develop a tool that works with multiple languages. The generated AST classes are never shared between languages, even if they share the definitions of the language constructs. Therefore, there are no interfaces that a generic tool could use. JastAdd would work well for defining AspectJava and AspectJLo, but there would be no IDE support. Defining AspectU would require more work because both the aspect weaver and the pointcut language would have to be reimplemented. In case of AspectU, this is a relatively large overhead.

Dinkelaker et al. present the POPART [5] meta-aspect protocol (MAP) on top of a meta-object protocol (MOP). The MAP extends the MOP such that it can intercept method calls, and adds support for aspect-orientation. Both POPART and Carpenter provide a generic aspect language that can be extended by creating subclasses, and both approach use a similar modularization of the weaving process. The key difference between both approaches is that POPART processes aspects at run-time, while Carpenter does that at compile-time. The MAP makes developing a language extension in POPART easier than in Carpenter, but the dependency on a MOP limits its applicability. Mainstream languages such Java, C#, and C++ do not natively support a MOP. To support JLo, its implementation would have to be rewritten as a dedicated virtual machine with MOP support instead a transformation to Java code. In addition, POPART cannot be used for the use case language, as it is not executable.

Dyer and Rajan [6] present *Nu*, an aspect-oriented intermediate language. The added *bind* and *remove* primitives add and destroy advising relations. The authors implemented *Nu* in the JVM and show that there is no significant performance impact, and demonstrate that *Nu* can model a wide range of aspect-oriented features. Similar to POPART, the approach is limited to executable languages, and requires modifications of the native implementations of mainstream languages. Because *Nu* reduces the gap between an aspect-oriented language and its execution environment, using *Nu* as a compilation target would significantly simplify our AspectJava weaver.

Haupt and Schippers [14] define a machine model for aspect-oriented programming. Aspect-oriented programming is modeled in a prototype based object-oriented language. Each object has a proxy that determines the identity of the object. Method calls are sent through a delegation chain with the proxy at the start and the object at the end. Class-based languages are supported by appending a shared class proxy to the chains for objects. Schippers et al. [31] demonstrate the expressiveness of the machine model by encoding four different languages. The machine model directly and elegantly models aspect-orientation instead of modifying join point shadows, which is the approach taken in Carpenter. The downside is that it would have to be implemented separately for each language run-time, which is not practical for mainstream languages. Carpenter does not depend on the implementation of a language run-time and is applicable to languages other than programming languages.

Tanter and Noyé propose Reflex, a kernel for multi-language aspect-oriented programming [35]. Their approach uses three layers. The first layer performs the actual weaving. The second layer manages aspect interactions. The third layer enables modular definitions of aspect languages. The authors use reified links to model the connection between advices and join point shadows. Interactions between aspects are resolved using link composition rules. To define an aspect language, a plugin is implemented which translates aspect programs written in that aspect language into a Reflex configuration. The authors implemented plugins for SOM and AspectJ. While Reflex AOP enables the definition of modular aspect languages, the host language is limited to Java. The principle is not Java-specific, but the kernel would have to be reimplemented for other host languages.

Heidenreich et al. present a model-driven approach to add modularization technique to a language [16]. They offer two ways of adding modularization to languages. The first approach is to extend the metamodel of the language by defining component interfaces. The second approach extracts those interfaces automatically. The latter technique has the advantage that existing tools keep working, but is sometimes more difficult to implement than a metamodel extension. They evaluate their approach by creating adding component capabilities to UML activity diagrams and the domain-specific language TaiPan. In earlier work, the authors have added aspect-orientation to Java [15]. The authors implemented the ReuseWare composition framework on top of the Eclipse Modeling Framework [4].

Weave.NET [21], Aspect.NET [30], and LOOM.NET [32] offer language-independent aspect-oriented programming by operating on the common language infrastructure (CLI) of the .NET platform. The use of CLI allows these aspect weavers to work with large collection of programming languages, and even support cross-language weaving. The latter is not supported by our framework, since there is no intermediate language that supports all possible languages. These approaches, however, reflect the object-oriented model behind the common intermediate language (CIL) in the aspect language that they define. While this is not a problem for object-oriented languages such as C# and Visual Basic, this is problematic for languages whose language constructs do not map well to object-oriented languages, such as functional and logical programming languages. The approach is also limited to programming languages.

SourceWeave.NET [18] uses an approach that is similar to that of the .NET approaches that operate on the CLI, but instead of operating on .NET assemblies, it operates on CodeDOM models. CodeDOM is the .NET standard for representing models of source code. Because CodeDOM is strongly related to CIL, SourceWeave.NET has the same limitations as the assembly based approaches.

Avgustinov et al. present abc, an extensible AspectJ compiler [1]. The abc compiler can use Polyglot [23] or JastAdd [7] for the front-end. A program is transformed to an intermediate representation called Jimple to perform the actual aspect weaving. The abc compiler uses a generic intermediate representation for pointcuts to simplify the development of new pointcut expressions. This representation is similar to the generic top layer of Carpenter. The abc compiler is limited to extensions of Java, but implements many optimizations to improve the performance of the woven program, which Carpenter does not do.

ALIA4J is an execution model for advanced-dispatching languages [2]. ALIA4J defines a language-independent metamodel for advanced-dispatching (LIAM), which is similar to the top layer of Carpenter. Concrete languages extend this model to define additional constructs. A plugin interacts with the JVM to ensure that the custom dispatching mechanism is used at run-time. An intermediate representation of a program is used to make the approach language-independent. The approach is not limited to building aspect-oriented extensions of languages, but because it uses run-time interception, it is limited to executable languages. The authors evaluate their approach by implementing language constructs from languages such as AspectJ and CaesarJ.

8. Conclusion

Aspect-orientation is added to ever more software engineering languages. Existing approaches to simplify the development of aspect-oriented language extensions are either limited in the types of supported host languages, or use complicated code generation techniques.

We defined Carpenter, an object-oriented framework for the development of aspect-oriented languages. Aspect-oriented languages constructs and the corresponding weavers are implemented directly in an object-oriented programming language. This approach enables the definition of abstractions that improve the language-independence of the aspect weavers without having to write a tool for generating weavers. Classes for generic aspect-oriented language constructs and generic weaving functionality can be reused from the Carpenter framework.

We used Carpenter to create aspect-oriented extensions of Java, JLo, and a language for use cases. This showed that a significant amount of work was saved by using Carpenter. Providing IDE support for aspect-oriented languages developed with Carpenter requires only a few lines of code.

Acknowledgements

We thank the reviewers for their insightful comments. We especially thank Eric Bodden for his support in the discussion of related work.

References

[1] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development I*, pages 293–334, 2006.

[2] C. Bockisch, A. Sewe, M. Mezini, and M. Akşit. An overview of ALIA4J. *Objects, Models, Components, Patterns*, pages 131–146, 2011.

[3] M. Brukman and A. Myers. PPG: a parser generator for extensible grammars, 2003. http://www.cs.cornell.edu/Projects/polyglot/ppg.html.

[4] F. Budinsky, S. Brodsky, and E. Merks. *Eclipse modeling framework*. 2003.

[5] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09*, pages 51–62.

[6] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08*, pages 191–202, 2008.

[7] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07*, pages 1–18.

[8] P. Fradet and M. Südholt. Towards a generic framework for aspect-oriented programming. In *Workshop on AOP '98, ECOOP*, pages 394–397, July 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. January 1995. ISBN 0201633612.

[10] A. Garcia, C. Chavez, T. Batista, C. Sant'anna, U. Kulesza, A. Rashid, and C. Lucena. On the modular representation of architectural aspects. In *Software Architecture*, volume 4344 of *LNCS*, pages 82–97. 2006.

[11] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04*, pages 36–45, 2004.

[12] R. Grimm. Better extensibility through modular syntax. In *PLDI '06*, pages 38–51.

[13] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03*, pages 60–69.

[14] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP '07*, pages 501–524.

[15] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect orientation for your language of choice. In *AOM at MoDELS'07*.

[16] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Software Development VI*, pages 39–82, 2009.

[17] S. Herrmann, C. Hundt, and K. Mehner. Mapping use case level aspects to Object Teams/Java. In *OOPSLA Workshop on Early Aspects*, 2004.

[18] A. Jackson and S. Clarke. SourceWeave.NET: Cross-language aspect-oriented programming. In *GPCE '04*, pages 115–135, 2004.

[19] I. Jacobson. Use cases and aspects–working seamlessly together. *Journal of Object Technology*, 2(4):7–28, 2003.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP '01*, pages 327–354.

[21] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03*, pages 1–12.

[22] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. *Programming Languages and Systems*, pages 105–121, 2003.

[23] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *CC '03*, pages 138–152.

[24] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. *ECOOP '05*, pages 214–240.

[25] K. Palma, Y. Eterovic, and J. M. Murillo. Extending the rapide adl to specify aspect oriented software architectures. In *15th International Conference on Software Engineering and Data Engineering*, page 170, 2006.

[26] T. Parr and R. Quong. ANTLR: A predicated (k) parser generator, 1995.

[27] J. Perez, E. Navarro, P. Letelier, and I. Ramos. A modelling proposal for aspect-oriented software architectures. In *IEEE International Symposium and Workshop on Engineering of Computer Based Systems '06*, pages 32–41.

[28] M. Pinto and L. Fuentes. AO-ADL: An ADL for describing aspect-oriented architectures. In *Early Aspects: Current Challenges and Future Directions*, volume 4765 of *LNCS*, 2007.

[29] S. Roychoudhury, J. Gray, and F. Jouault. A model-driven framework for aspect weaver construction. *Transactions on aspect-oriented software development VIII*, pages 1–45, 2011.

[30] V. Safonov and D. Grigoryev. Aspect.NET: aspect-oriented programming for Microsoft .NET in practice. *NET Developers Journal*, 7, 2005.

[31] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA '08*, pages 525–542, 2008.

[32] W. Schult, P. Tröger, and A. Polze. LOOM.NET-an aspect weaving tool. In *Workshop on AOP '03, ECOOP*.

[33] J. Sillito, C. Dutchyn, A. D. Eisenberg, and K. D. Volder. Use case level pointcuts. In *ECOOP '04*, 2004.

[34] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 53–60, 2002.

[35] É. Tanter and J. Noyé. A versatile kernel for multi-language aop. In *GPCE*, pages 173–188, 2005.

[36] É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In *Software Composition*, pages 227–242, 2006.

[37] M. van Dooren. Carpenter, 2011. http://www.cs.kuleuven.be/∼marko/carpenter.html.

[38] M. van Dooren and E. Steegmans. A higher abstraction level using first-class inheritance relations. In *ECOOP '07*, pages 425–449.