Creating a Modular AspectJ Foundation for Simple and Rapid Extension Implementation

by

Hristofor Mirchev

EWI

FMT

EXAMINATION COMMITTEE

dr. C.M. Bockisch

prof.dr.ir. M. Aksit

UNIVERSITY OF TWENTE.

30.09.2014

Abstract

The current state of aspect-oriented programming (AOP) has raised concerns regarding various limitations that AOP languages have. The issue is that AOP languages are not robust enough when the basis program is changed. There are many new proposals for AOP languages with new features that attempt to restrict or give more expressiveness to the programmer in order to force a new context where the problems can be mitigated. Some of those languages are designed as extensions of AspectJ. Existing open AspectJ compilers can be used for implementing such an extension, but this can quickly become a complicated task of extending the complex processes of lexing, parsing and weaving, of which the compilers offer low-level abstractions. Thus, there is a need for an easily extensible AspectJ foundation for a simpler and faster development of language extensions.

We have developed such a foundation and in this thesis we describe the design of the implementation. We provide an overview of a testing process to determine its validity. Finally, we implement one proposal for an AspectJ extension and evaluate the extensibility and ease of use of our foundation in comparison to other existing AspectJ compilers.

1. Introduction

Motivation. An easy approach to implement an aspect-oriented language would be to extend an existing AOP compiler. This is not always the best approach, however. Existing compilers may force the programmer to use or extend, components, that they rely on, which are not trivial to understand. They might have not been designed to conform to the same goals you seek or are simply not suited in the context of your design approach.

A simplified overview of the workflow of a compiler includes the following

sequence of operations. The source file is first subjected to a lexical analysis that breaks the stream of characters of the file into meaningful "chunks" (sometimes called *tokens*). The module that does this is called a *tokenizer* (or a *lexer*). After that, these tokens are processed according to the syntactical rules of the language, specified in a formal grammar. This part of the process is called *parsing* and the module that does that is called a *parser*. Finally, the parsed input is stored in a specific format to be interpreted and used to generate code. In AOP, for example, this is the step where the advice information would be woven into the original classes on a bytecode level in a complex process called *weaving*. Extending existing compilers requires some form of manual configuration to these weaving mechanics. This can prove to be a significantly inaccessible task for the average programmer. With our approach, we delegate the weaving process to the official AspectJ weaver, thereby freeing users of our implementation to focus more on the syntax of their language extension, rather than the complexity coming from handling the weaving mechanics. Splitting up the aspect information to be woven and the base Java statements is also accomplished in a clear and concise way through the use of Java annotations, which programmers are usually familiar with. These two factors alone reduce the complexity in using our implementation in comparison to existing AspectJ compilers.

For our implementation we carried out a *model-driven engineering* (MDE) approach. More specifically, we used the *EMFText* framework to develop a metamodel and grammar for AspectJ, which were used to generate an AspectJ parser. To handle the weaving process, we can delay it to the point that a class loader loads the class files and defines them to the *Java Virtual Machine* (JVM). This approach is called *load-time weaving.* Weaving in such a way can be done with the standard AspectJ load-time weaver, without the need for any user modifications. To accomplish this, however, we would need to "carry" the aspect

information to be woven in the .class file. We are doing this by transforming the aspect classes to the annotated style introduced in AspectJ 5. They can then be compiled with a regular Java compiler. The transformation is achieved through the use of a model-to-model transformation language.

Design Goals. Our approach is to implement a flexible AspectJ grammar with the following primary design goals:

- *simplicity* – we will measure simplicity by the number of concerns (e.g. lexing, parsing, weaving) that the programmer has modified over the lines of code (LOC) that he has written when implementing new functionality.
- *extensibility* – we will measure extensibility by the average number of artifacts (e.g. metamodel entities/AST nodes, syntax rules) that the programmer has reused when implementing new functionality.

In Section 3, where we explain our implementation of the AspectJ grammar in detail, we are further going to discuss the pre-existing AspectJ compilers and how they align with our goals and design process.

Contributions. The contributions of this thesis are the following:

- We have determined criteria for an easily modifiable implementation of an AspectJ compiler suitable for building research-oriented AspectJ extensions.
- We pose our implementation of an exstensible AspectJ grammar conforming to that criteria.
- We evaluate this implementation with a rigorous test process.
- We demonstrate the exstensibility of the implementation by presenting a grammar for an aspect-oriented programming language built on top of it.

- We evaluate our AspectJ foundation with respect to our design criteria by comparing our implementation of functionality from the extension with an implementation of the same functionality build on top of other existing AspectJ compilers.

Thesis Structure. The structure of the thesis is as follows. We first give an introduction of the necessary background information in Section 2, namely an overview of AspectJ, model-driven engineering and EMFText. In the following section we discuss in detail our implementation of an extensible AspectJ grammar. These two sections are written in collaboration with George de Heer since we both worked together on the implementation.

Next, in Section 4 we describe the test case we did, in which we implemented the Natural Aspects extension on top of that grammar, and evaluate, whether the design meets our initial criteria of extensibility. Finally, in Section 5 we draw some conclusions from the experience and discuss possible directions for future work.

2. Background

AspectJ. Aspect-oriented programming (AOP) is a paradigm that strives to increase modularity in programming by separating pieces of the program that rely on or affect other parts of it. While object-oriented programming (OOP) offers us a way to modularize common concerns, AOP offers a way to modularize cross-cutting concerns in particular.

AspectJ is a Java extension that supports AOP. It adds a few new concepts to Java. A *join point* is a point in the program flow. Examples of join points include method calls, method executions, constructor executions, object instantiations and others. A *pointcut* is a predicate on a set of join points that selects a subset of them. *Advice* is code meant to be implicitly executed when the program flow

encounters a join point matched by a pointcut. Finally, an *aspect* is the module that encapsulates all these new constructs. Another feature of AspectJ called *inter-type delcarations,* which affects the static structure of the program, namely it allows the programmer to add fields, methods or interfaces to existing classes, is outside the scope of our research [1].

Model-driven engineering. Model-driven engineering (MDE) is a system development methodology that focuses on the creation and use of *models*. Models are abstract representations of the concepts related to a specific problem domain. A model can describe the entities in a domain, their attributes, relationships between them and constraints on those relationships. Models are specified using some notation, described in a *modeling language*. This notation usually consists of at least a description of the *abstract syntax* (i.e. the concepts and relationships) and a description of the *concrete syntax* (i.e. the physical appearance of those concepts and relationships). The abstract syntax is commonly defined through a *metamodel*. All the terminology and considerations for models are applicable to the metamodel as well, as the metamodel is just a model of the model [2].

MDE strives to increase productivity in at least three respects. First, by raising the level of abstraction MDE closes the semantic gap for domain experts that may otherwise not be experienced in programming with a general-purpose language. Second, MDE tools can impose constraints and perform *model validity* checks to detect and prevent errors early in the development cycle. Model validity is the process of evaluating the model against different criteria, either coming from the metamodel or in the form of constraints, written by the programmer. Lastly, through the use of *code generation* and *model transformation*, MDE increases "automation" in software development thereby limiting the possibility of human errors. Code generation is the process of generating source code from the model

while model transformation is the act of transforming a source model into a target model through the use of transformation rules [3]. This is the primary reason why we find MDE to be particularly good at implementing a language conforming to our design goals. Most MDE tools generate a lot of the complex components (i.e. lexers and parsers) for the programmer, which leaves him with the task of implementing only a few highly modular elements (e.g. the metamodel of the language). Extensibility, on the other hand, can easily be achieved through the reuse of the metamodel or model transformations.

Frameworks for building applications and whole systems using models are called *modeling frameworks* or *language workbenches*. A very mature and popular modeling framework is the Eclipse Modeling Framework (EMF) [4]. Its metamodeling language (Ecore) is based on EMOF (a standardized metamodeling language) and it provides easy to use tools for code generation for EMF models that lay the grounds for interoperability with other EMF-based applications.

EMFText / JaMoPP. EMFText is a language workbench for defining textual languages, be it domain-specific (DSL) or general-purpose (e.g. Java) based on Ecore metamodels. It provides a rich DSL for syntax specification – the *Concrete Specification Language* (CS) based on EBNF, that can generate an editor with features like syntax highlighting and code folding, and components to parse and print instances of the metamodel [5]. The general development process with EMFText consists of the following steps [6]:

1. Specifying the abstract syntax for the language (the .ecore metamodel)

To define a language's metamodel we must consider how to break down the language into what *entities* it consists of, what *attributes* do they have and what are the relations or *references* between them. References have to further be distinguished into *containment* and *non-containment* references. A containment

reference relates an element of the model (a *parent*) with another, defined in the same context (a *child*). A non-containment reference relates a model element with one that is defined somewhere else. Let us consider the example of modeling a standard Java class. The class entity can hold a containment reference for a method declaration inside the class, but to model a method call statement one would use a non-containment reference to that method since it can be defined elsewhere.

2. Specifying the concrete syntax for the language (the .cs file)

Having completed the metamodel, we continue by defining the textual representation of all its entities in the .cs file. The .cs file can be roughly broken down to two sections. The first one contains metadata for the language. This can be information on what the file extension for the language is going to be, what is the root element of the language, what are the tokens (to help the lexer tokenise the input correctly) and how to highlight them, and some code-generation instructions. The second part of the .cs file contains the syntax rules for the language. A rule is a textual representation of a specific entity in the metamodel with its attributes and references. Various other elements such as keywords, operators for multiplicity (*, ?. +) and brackets for nested sub-rules are also regularly used in a syntax rule.

3. Generating the tools for the language

After defining the syntax specification, we can use the EMFText generator to create the accompanying language infrastructure. This includes the Java-based implementation of the metamodel, a parser and printer, reference resolvers that resolve names of non-containment references and classes related to an Eclipse-based editing functionality like syntax highlighting.

4. (Optional) Customizing the tools for the language

The previous step generates a basic tooling for the language. However, EMFText

offers ways in customizing it with additional advanced features like code completion, code folding, refactoring, semantic validation post parsing and more. For languages where the trivial reference resolvers, generated in the previous step, are not sufficient, EMFText also provides means for writing custom ones that override the behaviour. This can usually be the case in general-purpose programming languages where references can span cross-resources like Java or AspectJ.

The *Java Model Parser and Printer* (JaMoPP) is a complete implementation of Java 5 in EMFText. It offers a metamodel covering the whole language, a text syntax conforming to the Java specification and custom-written reference resolvers that correctly capture the Java static semantics when cross-referencing metaclass entities [7].

3. AspectJ Implementation

Before getting into the discussions of our AspectJ implementation in detail, let us examine the existing open AspectJ compilers and how they align with our design goals.

AspectBench Compiler. The AspectBench compiler (abc) is a complete implementation of the AspectJ 5 language that "aims to make it easy to implement both extensions and optimizations of the core language" [8]. It is based on the Polyglot [9] and Soot [10] frameworks.

A simplified overview of the workflow of abc begins when the Polyglot parser parses the input .java source file into an abstract syntax tree (AST). It then runs a series of transformations to separate the AST in two parts. One part holds only the pure Java constructs, while the other contains the additional AspectJ information like the advice bodies, inter-type declarations and others. The

process is then taken over by the Soot framework. It takes the purely Java AST and transforms it to its internal representation called *Jimple*. The framework then uses several modules that can convert freely between Jimples, Java byte code and Java source code to conduct the weaving process and output the final .class and .java files.

Using abc rather than our implementation has two disadvantages. First, Polyglot offers a clean and modular way to extend the grammar, but has a standard lexer for interpreting it, which is not extensible. This means that to make abc recognize the new language you build on top of it, one would need to copy and rewrite the existing lexer. This is not the case when using our approach, as the CS language in EMFText allows the reuse and extension of the defined lexing rules. Secondly, we argue that the weaving process in our approach is handled with less effort. The declarative way in which we "carry" the AspectJ information to be woven by using the annotation-based style is easier to understand than Polyglot's transformations that separate its AST. It also enables us to delegate the complexity of the weaving process to AspectJ's load time weaver, something that can not be done in abc, rather than having our users implement it themselves.

AspectJ-front. AspectJ-front is the combination of a syntax definition and a printer for AspectJ 5 and is made using the *Spoofax* modeling framework. The syntax definition is written in *SDF*, which is the metamodeling language in Spoofax. The printer is build in *Stratego/XT*. This is a subset of tools in Spoofax used specifically for program transformations. The printer is written as transformation rules that change the initial parse result (stored in Spoofax's *ATerm* format) into text.

AspectJ-front by itself is an extension of a similar combination of syntax

definition and printer for Java called *Java-front*. This already shows that extensions can be written with relative ease which matches one of our design criteria we set out to achieve - extensibility. AspectJ-front is also modular as the syntax definition is clearly separated from the printing rules. Thus, it also matches the second criteria we have set. The reason why we did not opt to use it is because it did not match the third design goal we have – simplicity.

We argue that picking EMF with respect to simplicity is better than the alternative of using Spoofax for two reasons. First, building an extension on top of previous work in EMF would require extending both the Ecore metamodel and the CS syntax definition. Having done that, EMF generates a parser and printer for you. The same result in Spoofax would require extending the SDF syntax definition, but also writing a new printer. One might argue that for the additional cost of writing the printer by hand, Spoofax at least skips the metamodeling step, however this is not truly the case. A programmer still has to have a mental image of the metamodel to follow when writing the syntax definition. EMF simply externalizes that process and produces a tangible artifact (i.e. the .ecore file) that can be shared and used as a specification between programmers. Secondly and more importantly, we believe that starting out with AspectJ-front in general is the harder approach. AspectJ-front is not mature and lacks proper documentation. The tool is outdated and barely supported anymore. Furthermore, when using our implementation, you do not have to implement any weaving logic, which is not the case with AspectJ-front.

ReflexBorg. "The ReflexBorg approach is a method for implementing aspect-oriented extensions of Java, including both their syntax and semantics" [11]. It consists of three layers. One layer is for the syntax definition of the language, written in SDF. Another is for the transformation of the abstract terms of the

aspect language into Java code instantiating Reflex elements, which is written in Stratego. The final one takes care of the semantics and weaving and is written in Reflex. Reflex is a Java implementation of a versatile kernel for aspect-oriented programming using bytecode transformation.

ReflexBorg uses the same metamodeling language (SDF) and the same model transformation tool (Stratego) as AspectJ-front, so the same concerns apply here as well.

The following subsections will explain the design of our AspectJ foundation by examining the metamodel we have developed, the transformation to annotation-based style and the testing process that we have used to evaluate it.

Metamodel. Following the naming convention set out from Kardelen, whose AspectJ prototype we used as inspiration, our metamodel consists of five subpacakges.

1. *Commons* package - contains entities for the top-level members in AspectJ (i.e. aspect, pointcut and advice).
2. *Pointcuts* package – contains entities for the 18 primitive pointcuts that the language supports.
3. *PcExp* package – contains entities for the acceptable pointcut combinators. Those are && (and), || (or) and ! (negated).
4. *Advice* package – contains entities for the 5 different advice types (i.e. Before, After, After Returning, After Throwing and Around).
5. *Patterns* package – contains entities for the pattern matching in pointcuts.

The Commons Package.

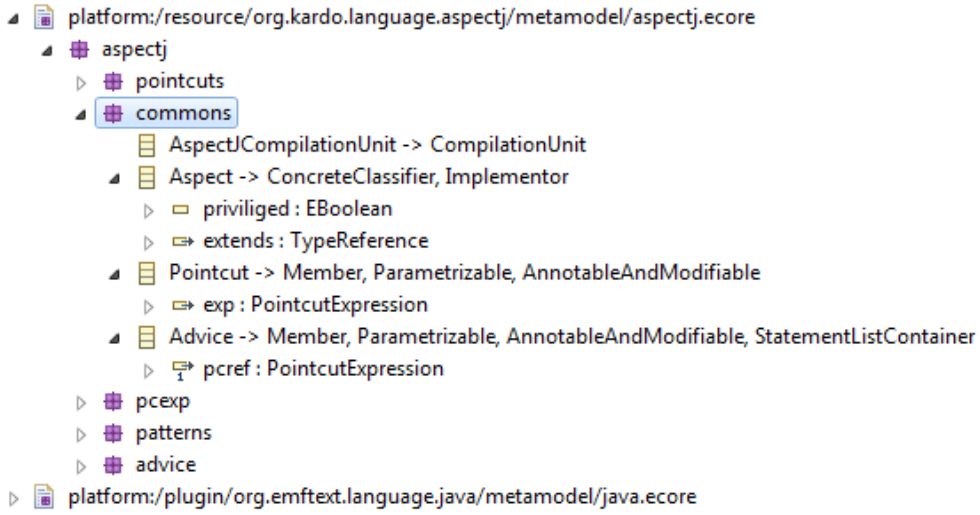

Figure 1. Overview of the Commons Package

The key design consideration here that drove the design of the whole package was whether an aspect should be treated "exactly" like a regular Java class. According to the AspectJ Developers Guide [1] an aspect declaration is syntactically similar to a class definition. Three of the differences they point out are that an aspect can cut across other types; that it can not be directly instantiated and that in case of nesting, the nested aspect must be static. However, the difference that influenced our metamodel design the most was that a class can not contain advice code.

According to the specification a regular class can contain pointcut definitions, but can not contain an advice block of any kind. This meant that there is a divide in the contents of an aspect (i.e. contents that can also be contained in classes and aspect-only code). This separation can be enforced in one of two ways. One way is to make a constrained metamodel that does not allow such mix-ups. Another approach is to make a more liberal metamodel that allows them, but write a post-processing semantic check that disregards such cases. We opted for the latter for two reasons.

First, it simplifies the metamodel. Enforcing such a constraint in the metamodel would result in an overhead of entities. We would need to make one entity that "captures" all the class contents (e.g. methods, fields and pointucts) and another for the Advice. This differentiation would also result in a need for two more entities so that a compilation unit can hold both aspects and classes. Without this overhead the metamodel is more streamlined.

Secondly, the JaMoPP model that we are using as a foundation comes with setter and getter methods implemented for the different entities it has. Directly extending these entities lets us reuse these methods which eased the process of implementing the custom reference resolvers for our AspectJ implementation.

Having settled on this issue the design became clear. The AspectJ compilation unit directly extends the Java compilation unit. Since it contains a reference to *ConcreteClassifiers,* we make the aspect extend ConcreteClassifier. We also directly extend *Implementor* to complete the functionality of an aspect to implement another. It contains one attribute of type boolean to determine if the aspect is priviliged or not and one reference to a type, in case the aspect extends another. The ConcreteClassifier entity contains a reference to *Member,* which is a supertype for the different class members (e.g. methods, fields). Thus, we model the pointcut and advice as Members. Both advice and pointcut entities contain a reference to a pointcut expression.

The Pointcuts Package.

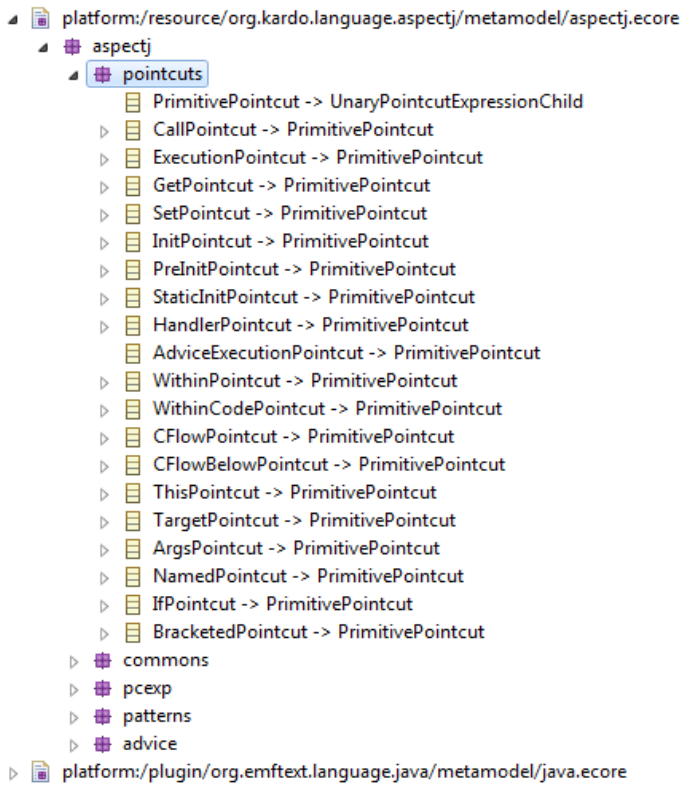

Figure 2. Overview of the Pointcuts Package

The package contains one entity for each of the 18 possible primitive pointcuts [1] and one common supertype *PrimitivePointcut* for easy polymorphic referencing. Each of the 18 entities contain a reference to the pattern they can match.

Such a design allows for an easy implementation of pointcut extensions. For a new type of primitive pointcuts, a programmer can add a new entity that extends PrimitivePointcut or he can modify it to introduce new global pointcut functionality.

The PcExp Package.

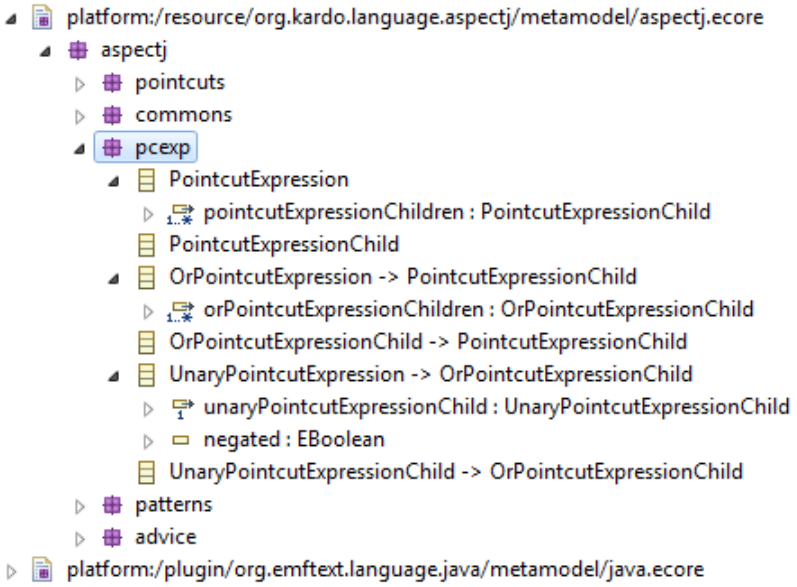

Figure 3. Overview of the PcExp Package

One tough problem that comes up when designing compound expressions is how to effortlessly implement the order of precedence of the operators. Our approach is inspired by the design of compound Java expressions in JaMoPP.

We have a *PointcutExpression* entity whose syntactical rule can be simplified as *<OrPointcutExpression> && <OrPointcutExpression>*, thereby giving least priority to &&. An OrPointcutExpression's rule on the other hand can be considered as *<UnaryPointcutExpression>* || *<UnaryPointcutExpression>*, which gives || second priority. Finally, an UnaryPointcutExpression is just a PrimitivePointcut that can either be negated or not, determined by a boolean attribute, giving ! top priority.

The Advice Package.

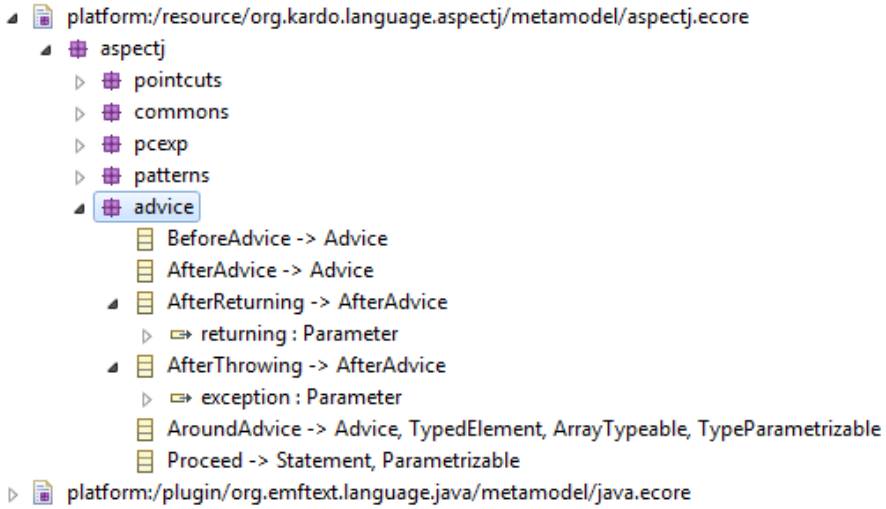

Figure 4. Overview of the Advice Package

The Before and After advice are simple and inherit directly from our basic Advice entity without extending it with any additional functionality. The After Returning and After Throwing extend the After advice with the addition of one extra reference to the returned or thrown parameter respectively. Finally, the return type of Around advice determines the need to inherit from *TypedElement*. *ArrayTypeable* and *TypeParametrizable* allow the return type to be an array or a generic respectively. We also cover the possibility for a *proceed* statement in the Around advice with the *Proceed* entity. The Proceed entity is a direct subtype of *Statement* which means that, according to the metamodel, a call to proceed is valid from every type of advice or any other Statement container. Although this is wrong, we again chose to have a more liberal metamodel and introduce the constraints in a post-processing step in the future.

The Patterns Package.

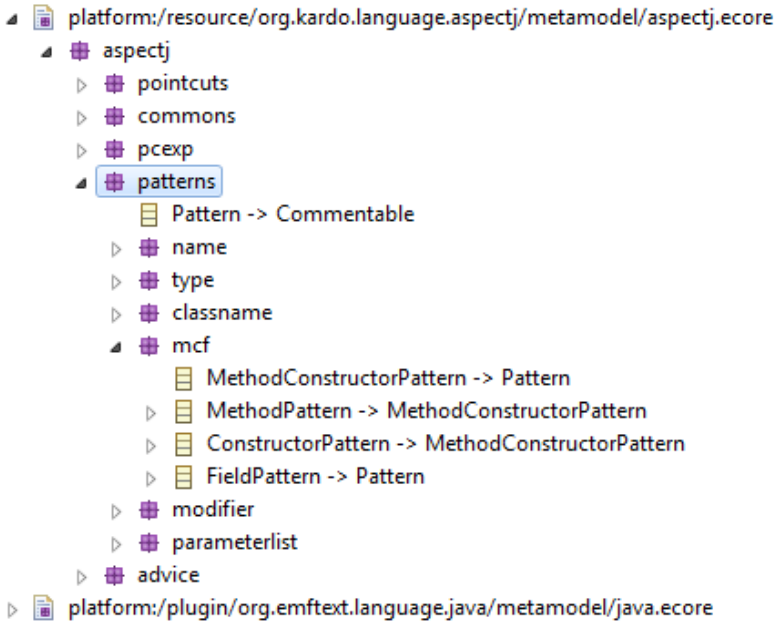

Figure 5. Overview of the Patterns Package

Due to the numerous symbols and combinations, and possible ambiguity, implementing the pattern matching mechanics proved to be one of the hardest parts of designing the metamodel. We used the abc AspectJ grammar [12] as a reference guide for this particular part of the process.

There are three important families of patterns. The first one is located in the *mcf* package. It contains the *FieldPattern*, the *ConstructorPattern* and the *MethodPattern*. The first one is used in case of a *get* and *set* pointcuts. The second is used in the context of *initialization* and *preinitialization* pointcuts. The final one and the ConstructorPattern are subtypes of the *MethodConstructorPattern* since many of the pointcuts can match either a method call or a constructor call (e.g. *call*, *execution*, *withincode*). The second family of patterns match class names. They are located in the *classname* package and are used with respect to the *within*, *handler* and *staticinitialization* pointcuts. The final big family of patterns is

responsible for matching types and identifiers and is located in the *type* package. Patterns from this family are used with *this*, *target* and *args* pointcuts.

The rest of the patterns serve as parts of or helpers to the patterns from these three families.

Transformation. AspectJ 5 introduced an alternative to writing aspect declarations in the traditional way by incorporating a new annotation-based style. The gained benefit is that programs written in this annotation style can now be compiled with a regular Java 5 compiler and be woven separately at a later stage. Typically both of these processes are done at compile-time by the ajc compiler. Separating these two concerns (compiling and weaving) is what allows us to focus on making a moular AspectJ implementation and handle the compiling and weaving processes with traditional tools (i.e. the Java compiler *javac* and AspectJ's load-time weaver). Unlike in other AspectJ compilers like abc, not having to implement complex weaving mechanics is the key factor for an easier and faster extension development.

The following table provides an overview of the main AspectJ concepts written in both regular and annotation-based styles. The examples were taken from the official *AspectJ 5 Development Kit Developer's Notebook [1]*.

Regular style	Annotation style
public aspect <Foo> { }	@Aspect **public class** <Foo> { }
pointcut <AnyCall>() : <**call**(* *.* (..))>;	@Pointcut("<call(* *.* (..))>") **void** <AnyCall> () { }
before() : <**call**(* *.*(..))> { }	@Before("<call(* *.*(..))>") **public void** bfAdvice () { }
after() : <**call**(* *.*(..))> { }	@After("<call(* *.*(..))>") **public void** afAdvice () { }
after() **returning**(Foo <f>) : <**call**(* *.*(..))> { }	@AfterReturning(**pointcut**="<call(* *.*(..))>", **returning**="<f>") **public void** afrAdvice (Foo <f>) { }
after() **throwing**(Exception <e>) : <**call**(* *.*(..))> { }	@AfterThrowing(**pointcut**="<call(* *.*(..))>", **throwing**="<e>") **public void** aftAdvice (Exception <e>) { }

Object around(int <i>): <setAge(i)> {
return proceed();
}
	@Around("<setAge(i)>")
	public Object arAdvice
	(ProceedingJoinPoint jp, int <i>) {
	return jp.proceed();
	}

Table 1. AspectJ Components in Regular and Annotation Style

To get the model we obtain after parsing from the regular AspectJ style to the annotation style, we must use a model transformation language. More precisely, we need a model-to-model transformation language to translate a model conforming to our AspectJ metamodel to one that conforms to the Java metamodel provided by JaMoPP. We considered two of the most popular and

mature transformation frameworks that support EMF-based models – *Model-to-Model Transformation* (MMT) [13] and *Epsilon [14]*.

MMT consists of two very distinct model-to-model toolkits – *QVT* and *ATL*. QVT is a standardized set of three model-to-model languages – QVT-Operational, QVT-Relations and QVT-Core. The first one is an imperative language, while the other two are both declarative and are therefore commonly jointly called QVT-Declarative [15].

ATL was initially designed as an alternative to QVT before getting paired with it in MMT. ATL supports both imperative and declarative styles of writing transformations. The recommended style is declarative as it is better for simple and straightforward transformation rules, but imperative can also be used for more complex ones [16].

Epsilon is a rich toolset that can be used for model validity, model comparison, code generation and model-to-model transformation. The framework provides a language for each of those functionalities. All of those languages, however, are minimal extensions built on top of a common imperative language – the *Epsilon Object Language* (EOL). The language for the model-to-model transformation is called *Epsilon Transformation Language* (ETL). Like ATL it is a hybrid language in the sense that it supports both imperative and declarative styles of writing [14].

In comparison, both MMT and Epsilon allow us to have rules that transform any number of input models to any number of output ones. Both frameworks also support imperative and declarative styles of writing. Finally, both frameworks are mature and rich, and offer a diverse set of extra functionality like syntax highlighting, error detection and debugging in Eclipse. With respect to these common classification criteria Epsilon and MMT are similar to each other. The only deciding requirement we had was how easy it is to call class methods outside the context of the transformation, since, as can be seen from Table 1, to transform

a pointcut or advice a programmer would need to transform the pointcut expression they contain to a string and pass it along as a parameter of the annotation. Since no transformation language would be able to perform this task natively, we needed an easy way to call our generated printer to do that.

In QVT such cases are referred to as "black box operations" [17]. We found this approach to make the project and its design more complicated. Epsilon, on the other hand, has a clear way of doing this and even lists it among the main features to use the framework [14]. Later in this section we are going to explain more in-depth exactly how we accomplished this task as it proved to be rather challenging, but this was the sole reason we picked Epsilon over MMT. The choice, however, will also allow us to re-use code from our transformation rules to implement model validity or unit tests for the transformation in future work, as all languages in Epsilon share a common syntactical foundation.

ETL transformations are organized in a module that can contain an arbitrary number of uniquely named transformation rules. As well as transformation rules, an ETL module can optionally contain any number of *pre* or *post* blocks of statements which are executed before or after the transformation respectfully [18]. The following listing displays the syntax for a transformation rule and the post/pre blocks.

```
1.  (pre | post) <name> {
2.    statements+
3.  }
4.
5.  (@abstract)?
6.  (@lazy)?
7.  (@primary)?
8.  rule <name>
9.    transform <sourceParameterName> : <sourceParameterType>
10.   (, <sourceParameterName> : <sourceParameterType>)*
11.   to <rightParameterName> : <rightParameterType>
12.   (, <rightParameterName> : <rightParameterType>)*
13.   (extends <ruleName> (, <ruleName>)*)? {
14.
15.   (guard (:expression) | ( { statementBlock }))?
16.
17.   statements+
18. }
```

Listing 1. Syntax of a Transformation Rule and Pre/Post block

The pre and post blocks consist of the respective identifiers (*pre* or *post*), an optional name for the block and the set of statements to be executed. The transformation rule can be declared as *abstract*, *lazy* or *primary* via annotations, followed by the *rule* identifier and the rule name. The source and target models are declared following the *transform* and *to* keywords. A rule can also extend any number of different transformation rules declared after the *extends* keyword. Apart from the EOL statements a programmer can also specify a guard statement to limit the applicability of the rule to a selected subset of source models [18].

In the following subsections we are only going to demonstrate our transformation rule for the *Before* advice as the rest are analogous. We believe it

still sufficiently captures most of the challenging logic we faced when designing the transformation.

As can be seen from Table 1, a Before advice declaration using the annotation style is just a regular public Java method of type *void* that has the *@Before* annotation. Two important considerations here are:

1. Although the advice declaration does not have a name, the Java method must have an unique name.
2. The pointcut expression that the advice contains is passed as a string parameter to the annotation.

Listing 2 shows our transformation rule for the Before advice.

```
1.  rule Advice2Method
2.    transform ajAdvice : aspectj!Advice
3.     to jMethod : java!ClassMethod {
4.
5.    jMethod.name = getUniqueAdviceName(ajAdvice);
6.    jMethod.parameters = ajAdvice.parameters;
7.  }
8.
9.  rule BeforeAdvice2Method
10.   transform ajBeforeAdvice : aspectj!BeforeAdvice
11.    to jMethod : java!ClassMethod
12.    extends Advice2Method {
13.
14.    jMethod.annotationsAndModifiers.add(getAnnotation(ajLibBefore!
              Commentable.allInstances().first(), "Before", ajBeforeAdvice.pcref));
15.    jMethod.annotationsAndModifiers.add(new java!Public);
16.
17.    jMethod.typeReference = new java!Void;
18.    jMethod.statements = ajBeforeAdvice.statements;
19. }
```

Listing 2. Before Advice Transformation Rule

Lines 1 – 7 describe a common rule that all other advice rules extend. It takes care of the first consideration we noted by calling a *getUniqueAdviceName* helper method and passes along the parameters that the advice might have as parameters of the new method. Lines 15, 17, 18 set the method to be public, be of type *void* and pass along the body of statements the advice contains as the body of the new method. These three lines are common for all advice types except *Around* (the annotated method for Around has the return type of the Around advice itself and a *proceed* statement will get transformed rather than copied verbatim). Thus, one can argue that those three lines could also be extracted to the common advice rule and have a separate and specific rule for Around. We liked our approach better as an Around advice is still a type of advice and thus the relationship is still of an *is-a* kind, which is best represented by inheritance. Finally, line 14 and the call to *getAnnotation* handle the second consideration we mentioned.

Let us now demonstrate the implementation of these two methods: getUniqueAdviceName and getAnnotation.

```
1.  pre {
2.      var globalAdviceCounter : Integer = 0;
3.  }
4.
5.  operation Any getUniqueAdviceName(ajAdvice : aspectj!Advice) : String {
6.      globalAdviceCounter = globalAdviceCounter + 1;
7.      var name : String = "aj$" + ajAdvice.eClass.name + "$" +
                ajAdvice.eContainer.name + "$" + globalAdviceCounter + "$" +
                ajAdvice.hashCode();
8.
9.      return name.replace(" ", "_");
10. }
```

Listing 3. Getting the Unique Name for Advice Methods

Listing 3 shows how we handle the first consideration of generating an unique name for the advice method. We looked at how the official AspectJ compiler handled the same issue and tried to imitate the same behavior. Line 7 shows how we form the name of the method. We start with the string *aj* (a mnemonic for AspectJ), concatenate the type of advice (in our example this would be "Before"), the name of the aspect that contains the advice, a global counter of advice and finally add a hash value. We use a pre block to create an advice counter variable as since it gets executed only once before the actual transformation, it simulates the global variable we need.

```
1.  operation Any printModelElement(elem : Any) : String {
2.    var resourcePrinter = new
    Native("org.kardo.language.aspectj.resource.aspectj.mopp.ParameterlessAspectj
    Printer");
3.
4.    if (elem.isDefined()) {
5.      return resourcePrinter.printElement(elem);
6.    } else {
7.      return "";
8.    }
9.  }
10. …
11. operation Any getAnnotation(libModel : Any, annotation : String,
    pointcutExpression : aspectj!PointcutExpression) : java!AnnotationInstance {
12.   var anno = makeAnnotationInstance(libModel, annotation);
13.
14.   if (pointcutExpression.isDefined()) {
15.     var pcExp : String = printModelElement(pointcutExpression);
16.     var param = new java!SingleAnnotationParameter;
17.     param.value = getParameterValue(pcExp);
18.     anno.parameter = param;
19.   }
20.   return anno;
21. }
```

Listing 4. Getting the Advice Annotation with the Pointcut Expression

Creating and setting up the annotations was one of the major issues we faced while working on the transformation. There are two problems with this task.

First, how to create an instance of the annotation entity of the Java metamodel we use that "points" to the actual annotation located in the AspectJ library. An annotation in JaMoPP's Java metamodel is represented by the *AnnotationInstance* entity. It contains an optional namespace part (i.e. an ordered set of strings) for

the fully qualified annotation name, and a name part (i.e. an element of type *Classifier*). Classifier is another entity in the metamodel, but it is abstract, which means we can not directly construct it and set it to the correct annotation name (i.e. Before in our case). What we have to do is find the correct concrete subtype of Classifier and then assign the annotation name to it. To solve the problem we took all the actual .java files of the annotations from the AspectJ library, parsed them with JaMoPP and fed the resulting models to the transformation. At that point, setting the name part of the annotation we are trying to create was just a mapping to the correct model entity. The query to obtain the correct model is done in the *makeAnnotationInstance* method.

The second problem we faced was how to get a string representation of the pointcut expression that advice contain. Lines 1 – 9 of Listing 4 demonstrate our solution. The easiest way to accomplish this task is to call the generated printer for our AspectJ model and pass the pointcut expression. ETL offers an easy way to access a class outside the context of the transformation, however the only requirement is that the class has a no-argument constructor. Since the generated printer class does not have such a constructor, we decided to write a custom printer class that extends the original and write such a constructor for it. We prefer this over the alternative of simply adding such a constructor in the generated printer, as this way is more extensible and imposes a clear separation of the generated and non-generated code. On line 2 we create a variable that points to the custom printer and on line 5 we call its *printElement* method to obtain the string representation of the pointcut expression.

Testing and Evaluation. In the previous subsections we presented our AspectJ metamodel as well as some challenges and design decisions we faced along the way. In order to evaluate the approach we have devised a test suite to

demonstrate the correct parsing of AspectJ applications in reference to the official AspectJ compiler - ajc.

The goals of the suite are to test 1) that our parser accepts valid AspectJ code, 2) that the model instance created after parsing has the expected structure, 3) that the generated printer outputs a code representation of the model that is semantically equivalent to the input. To achieve these goals we employed a testing process similar to how JaMoPP was tested [7]. The process is shown on Fig. 6.

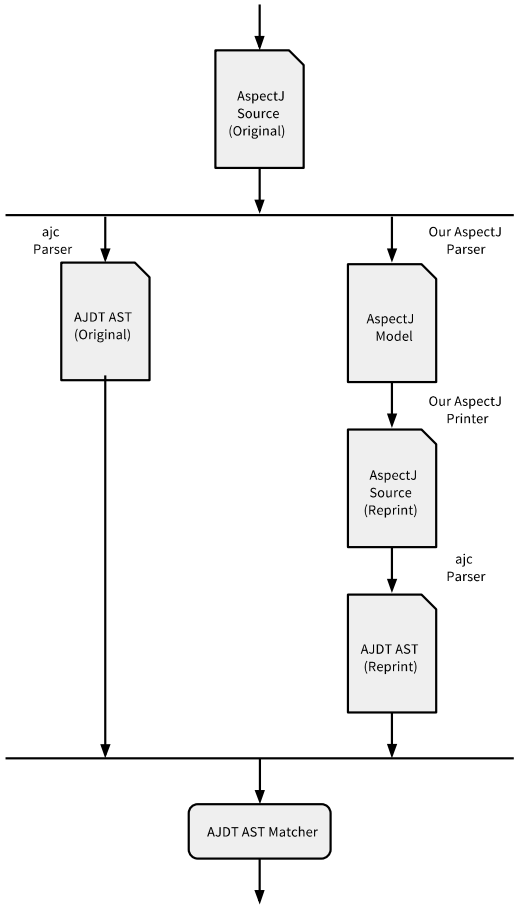

Figure 6. Test Process for the AspectJ Implementation

Beginning with a valid AspectJ source file, both our generated parser and the

reference one (the ajc) process the file and create their respective internal representations of it. In our case, this is a model instance of the AspectJ metamodel with unresolved cross-references. In ajc's case, this is an abstract syntax tree (AST). Next, our reference resolvers attempt to resolve the cross-references after which the generated printer reprints the model in its text form. The reprinted source file is fed to the reference compiler which creates another AST. Finally, the AST of the reprint is compared to the original one via an AST matcher provided by ajc.

We believe this approach meets the goals we set out to achieve. First, if our parser accepts all the valid AspectJ test programs we give it and not throw a parsing error, then we can conclude that the soundness property in our first goal is met. Secondly, the structure of the model instance is checked for correctness by the AST matcher. If there are any unresolved or missing elements, they will cause resolving errors and not get reprinted, which will be detected by the AST matcher. Lastly, any other mismatches that might raise error messages will also be detected by the matcher. In those cases, we manually checked the reprinted source file and compared it with the original to discover the source of the error. In nearly half of our test cases we exhibited such reprinting errors although all of the test files were parsed without errors. Often times white space, empty blocks and other layout information led to the mismatches causing the errors.

Due to limitations we imposed on the metamodel by design, running our test suite with official AspectJ benchmarks was impractical as it would require us to go through each source file and modify uses of functionality we did not implement. Thus, as input for the test process we provided 18 AspectJ files we custom wrote ourselves. We tried to achieve maximum coverage by writing test cases for the different variations of each element in our AspectJ model following the official *AspetJ 5 Quick Reference [19]*. The overview of the test files separated over the

packages in our metamodel is the following:

- *Advice* package – Contains 5 test files, one for each advice type. Each file tests for an empty advice, for an advice that exposes a parameter, for a *strictfp* advice and for an advice written in annotation style.
- *Commons* package – Contains 3 test files. The first one test for possible classifier declarations in a compilation unit (i.e. an aspect, a class, an annotation, an enum and an interface). The second tests for possible kinds of non-AOP contents of an aspect (i.e. a nested class, a field and a regular method). The last one tests for possible pointcut declaration variations (i.e. one without a modifier, one with a modifier, one without a pointcut expression, one where the pointcut expression is just a primitive pointcut and one where it is a conjunction).
- *Patterns* package – Contains 8 test files that also thoroughly test the variations of the pattern types we have modeled (refer to the patterns subsection in Sec. 3).
- *Pcexp* package – Contains 1 test file. It checks for a pointcut declaration with a regular pointcut expression and one with a negated pointcut expression.
- *Pointcuts* package – Contains 1 test file. The file contains 18 pointcut declarations that test out possible primitive pointcuts (e.g. call, execution).

The testing was automated with the Junit framework and special effort went into making the environment easily extendable with more tests. A programmer simply has to put a valid AspectJ source file ending with the .aspectj extension of our implementation in the *src-input* folder of the testing project and then rerun the tests. In the end all 18 files passed the test suite.

In conclusion, we can say that while this test process does not guarantee completeness, it does give us enough confidence that our AspectJ implementation

can be the foundation of language extensions that can be used in practice.

To test the transformation we fed the 18 files we had written to our parser and ran the transformation on the results. After that, we manually ran through the 18 outputted models and checked if they have been transformed according to the transformation rules in the *AspectJ 5 Development Kit Developer's Notebook [1]* (part of those rules you can see in Table 1).

Apart from differences in layout information, all the resulting models satisfied our expectations with respect to the official guide. We can, therefore, say that the percent of the metamodel covered by these 18 test files is also transformed correctly.

A better testing approach would be to write a script to run all the test files we have through our parser. After that, we would execute the transformation on the resulting models and run them through the JaMoPP printer. The resulting Java source files can then be put through a testing process similar to the one we used for the metamodel. The difference being that this time we use JaMoPP's parser, the standard Java compiler javac and the JDT AST matcher rather than their respective AspectJ equivalents. Setting up such a testing environment would take much time in integrating JaMoPP, javac and the JDT matcher, so we left this for future work and opted for a smaller and more manual approach.

4. Natural Aspects Implementation

Natural Aspects [20] is a proposal for a language extension of AspectJ that introduces a minimal amount of additions that make for a more natural style of aspect-oriented programming. The language allows the declaration of *events*. Event declarations are similar to aspect declarations and collect context information in local fields over time. Events are also side-effect free. Another

addition to AspectJ are the *event detectors*. Event detectors are named pointcut expressions that can also contain a reference to a named event or event detector. The language calls the pair of (event detector, response) a *basic unit*. One such unit that is introduced is the *when* basic unit. It can contain a special *trigger* operation that announces events upon their detection. Finally, the language proposes a method for aspect composition through the use of a *composes* clause and several declare statements that enforce a strict ordering of the composed aspects.

Natural Aspects strives to achieve three main goals: (1) complete separation of event identification from response; (2) natural composition of both events and aspects; and (3) loosen the coupling between the aspects and the base program.

Metamodel. The structure of the Natural Aspects metamodel contains four subpackages:

1. *Commons* package – contains entities for the top-level members in Natural Aspects (i.e. an event, an event detector and the new extended aspect).
2. *BasicUnits* package – contains entities for the *when* basic unit and the accompanying *trigger* statement.
3. *Composition* package – contains entities for the composition of aspects and the *local declare* statements that determine the ordering in those cases.
4. *EventDetectors* package – contains entities for the new primitive pointcuts (i.e. those that refer to a named event and a named event detector).

The Commons Package.

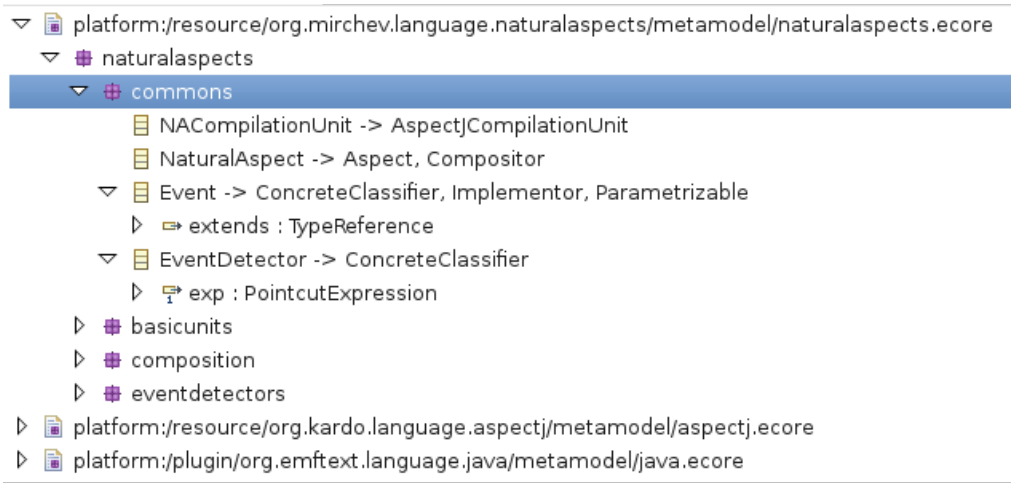

Figure 7. Overview of the Commons Package

The compilation unit directly reuses the AspectJ compilation unit without extending it with any functionality.

The *NaturalAspect* is the new entity that models the concept of aspects in the language. The main change to aspects that the language introduces is the ability of an aspect to be defined as a composition of other aspects. The new entity reuses the old one from the AspectJ implementation and, in addition, inherits from *Compositor,* which is a new entity introduced in this extension. The role of the Compositor entity is to be an interface from which you inherit aspect composition behavior.

The *Event* entity models the new concept of events. The structure of the Event entity is similar to that of an aspect (i.e. it inherits from the same supertypes - ConcreteClassifier and Implementor) with the addition of *Parametrizable* to account for the parameters that an event can expose.

Finally, the *EventDetector* represents the concept of event detectors in the language. Event detectors are named pointcut expressions that can be written outside the scope of an aspect declaration. To make event detectors on the same structural level as aspects and events, they must also extend ConcreteClassifier.

The BasicUnits Package.

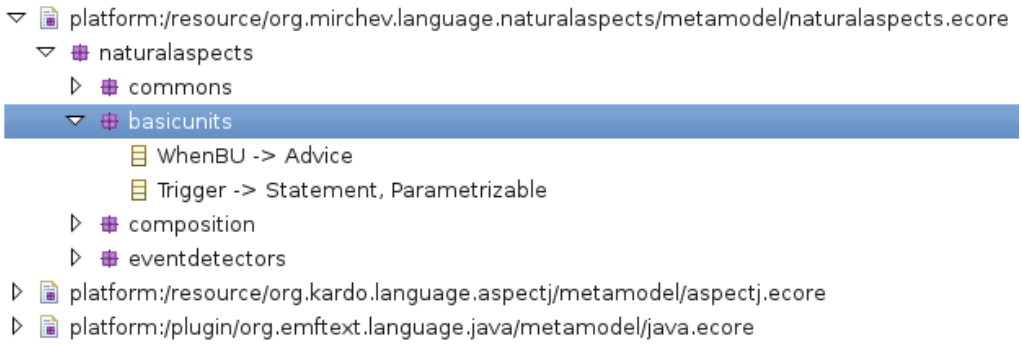

Figure 8. Overview of the BasicUnits Package

The *WhenBU* entity is a direct extension of Advice as basic units are practically renamed blocks of advice. Similarly to the Proceed entity from the AspectJ foundation, *Trigger* is a direct subtype of Statement which will result in a valid model from an invalid piece of code (i.e. if the trigger call happens from an advice not of type *when*). However, the same consideration there still holds. Making a more restrictive metamodel will result in an overhead of copied entities for every kind of statement, while leaving it more liberal, with a post-processing semantic check, results in a simpler and organized metamodel.

The Composition Package.

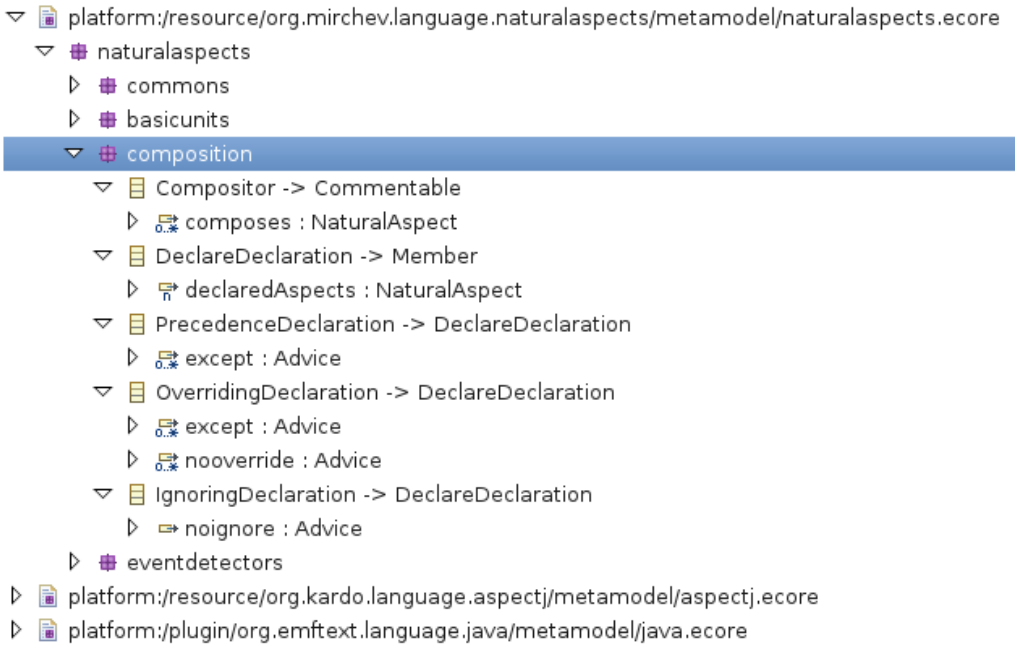

Figure 9. Overview of the Composition Package

With the introduction of aspect composition (achievable by extending the Compositor interface), the language offers a few ways of controlling the ordering in the compound aspect. This is done via the use of *declare* statements.

There are three declare statements. Let us demonstrate their functionality with an easy example consisting of one aspect A, that is composed of aspects B and C. With the *local declare precedence B, C* statement all basic units in B will precede those in C. An optional keyword *except* can denote a list of pairs of basic units whose order is the reverse of that stated in the declare statement. The *local declare overriding B, C* means that if basic units from B and C are applicable at a certain join point, only those from B will respond. In addition to the except clause, this statement offers an optional *nooveride* list for basic units that do not override each other. Finally, the *local declare ignoring B, C* means that responses from basic units in B will not trigger if the join point was matched by an event detector in C.

Similar to the nooveride clause, this statement provides a *noignore* one that overrides this behavior.

In the metamodel all three of the local declare statements share a common supertype *DeclareDeclaration* that contains a reference to the aspects mentioned in the statement. The concrete subtypes for each of the three (*PrecedenceDeclaration*, *OverridingDeclaration* and *IgnoringDeclaration*) contain references for the exclusive optional clauses.

The EventDetectors Package.

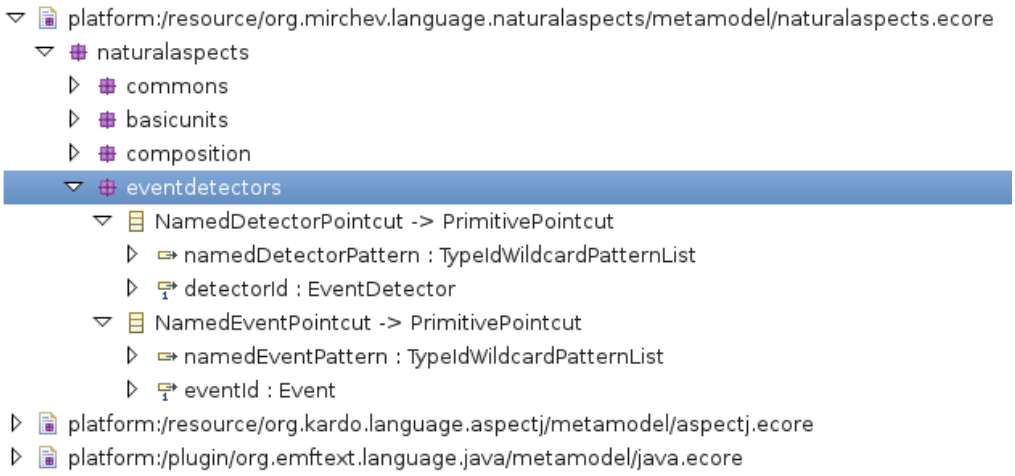

Figure 10. Overview of the EventDetectors Package

The Natural Aspects paper [20] allows event detectors to contain pointcuts that refer to named events and named event detectors. To accomplish this we have extended the PrimitivePointcut entity from our AspectJ implementation with two new pointcut types. The first one is responsible for the named events (*NamedEventPointcut*) and contains a reference to an event. The other is accountable for the event detectors (*NamedDetectorPointcut*) holding a reference to one.

Transformation. The following table gives an overview of the new concepts introduced in Natural Aspects and how they translate to AspectJ. The examples were taken from the paper that proposed the extension and modified for brevity [20].

	Natural Aspects	AspectJ
When Basic Unit	**when**(P <product>): <**call**(P.timeDone())> { }	**Object around**(P <product>): <**call**(P.timeDone())> { }
Event	**event** <LowActivity>(P product) { }	**aspect** <LowActivity> { }
Aspect	**aspect** <LowActivityDiscount> { }	**aspect** <LowActivityDiscount> { }

Event Detector	<FinishedProduct>:
	<**call**(P.timeDone())>
		aspect <FinishedProduct> {
		pointcut <FinishedProduct>Pointcut:
		<**call**(P.timeDone())>;
		}

Table 2: Natural Aspects Components Translated to AspectJ

The when basic unit is transformed to an around advice with the same parameters, pointcut expression and statements. Special treatment is needed in case of a trigger statement in the basic unit. We are going to explain this scenario more in-depth in the following subsections as it was the main challenge we faced with this transformation.

The event is transformed into an aspect and all its attributes are copied verbatim.

A simple aspect (one that is not composed from other aspects) is copied verbatim without any modifications. In case of a compound aspect we modify the statements that it contains. We are going to use the following code snippet to illustrate the four cases for that modification depending on the three declare

statements.

```
1.  aspect A {
2.   BasicUnitA:
3.    after(): pcA() { }
4.  }
5.
6.  aspect B {
7.   BasicUnitB:
8.    before(): pcB() { }
9.  }
10.
11. aspect C composes A, B { }
```

Listing 5. Basic Aspect Composition Example Scenario

1. No declare statements – In this case we copy *BasicUnitA* and *BasicUnitB* (lines 2-3, 7-8) and add them to the contents of aspect C.
2. A *local declare precedence A, B* statement – In this case we also copy the two basic units, only this time the order to add them is specific (i.e. *BasicUnitA* must be added before *BasicUnitB*).
3. A *local declare overriding A.BasicUnitA, B.BasicUnitB* statement – In this case we copy *BasicUnitA* and add it to the contents of aspect C. We proceed to copy *BasicUnitB* but before adding it to C, we modify the pointcut expression of the basic unit to *pcB( ) & !pcA( )*. This is to achieve the intended behavior to force the new aspect to disregard join points that get matched by both basic units and only respond to those that get matched by either one, or the other.
4. A *local declare ignoring A.BasicUnitA, B.BasicUnitB* statement – In this case we take an approach similar to the last one. Only this time we

modify the pointcut expression of *BasicUnitB* to *pcB( ) & !cflow(pcA( ))*. Again, this is to achieve the intended behavior of not triggering the response of the aspect if joint points that are in the workflow of *pcA( )* get matched by *pcB( )*.

Since the event detector is a top-class entity that can be written outside of the scope of an aspect and in AspectJ the pointcut expression must be in one, we transform the event detector to an empty aspect and add a new pointcut with the event detector's pointcut expression.

Handling the trigger statement in a when basic unit proved to be the hardest challenge while doing this transformation. There are several considerations when mapping an event declaration with a trigger statement to pure AspectJ syntax.

- We have to be careful not to introduce external side-effects as per design the event declarations should go without.
- We should have a way of determining if the event was triggered that is thread-safe.
- We should have a public interface to refer to the event in case of an event detector with a reference to that event.

The following listings demonstrate our approach in handling these problems.

```
1.  rule WhenBU2AroundAdvice
2.    transform naWhenBU : naturalaspects!WhenBU
3.    to ajAroundAdvice : aspectj!AroundAdvice {
4.
5.    …
6.
    ajAroundAdvice.statements.addAll(getWhenStatements(naWhenBU.statements));
7.    ...
8.  }
9.
10. operation Any getWhenStatements(statements : Any) : Collection {
11.   var result = new OrderedSet;
12.
13.   for (statement : java!Statement in statements) {
14.     if (statement.isTypeOf(naturalaspects!Trigger)) {
15.       result = handleTrigger(result, statement);
16.     } else {
17.       result.add(statement);
18.     }
19.   }
20.   return result;
21. }
```

Listing 6. When Basic Unit Transformation Rule

Line 6 of Listing 6 passes the statements of the basic unit to the *getWhenStatements* method which checks every statement whether it is a trigger statement, in which case it passes it along to the *handleTrigger* function, or otherwise adds it to the resulting collection. Let us now demonstrate the implementation of the *handleTrigger* method.

```
1.  operation Any handleTrigger(col : Any, statement: Any) : Collection {
2.      var result = col;
3.      var containingEvent = statement.eContainer.eContainer;
4.
5.      //ThreadLocal Stack
6.      var stack = getThreadLocalStack();
7.      addStackToContainingEvent(stack, containingEvent);
8.
9.      //Public Pointcut as Interface to Event
10.     var pointcut = getPointcutAsInterface(statement.eContainer.pcref, stack);
11.     containingEvent.members.add(pointcut);
12.
13.     //Side-Effect Free Populating of the Stack
14.     result.addAll(populateStack(stack));
15.
16.     return result;
17. }
```

Listing 7. Handling the Trigger Statement

The method consists of three steps that address the three concerns mentioned before.

First, we need a way to determine if the event is triggered. Since this can be accomplished from several different points in the program flow and multiple aspects could respond in different ways to it, a simple Boolean flag to mark if the event was triggered would not be enough. We have to use a thread-safe data structure for the possibility of multiple triggering. On line 6 we call the method *getThreadLocalStack* to create a thread-local instance of a stack and on the following line add it to the event that contains the when basic unit that we are currently transforming.

Secondly, we need to populate the stack in a way that is free of external side-

effects. This is done on line 14 by calling the *populateStack* method. What it does is to push a *True* object in the stack that marks that we have passed a trigger statement, crates an Object on which to pass the result of a call to *proceed( )*, after which pops the top value of the stack and returns the created Object with the proceed workflow.

Finally, we need to set up a public pointcut as an interface to other elements. This is achieved by calling the *getPointcutAsInterface* method on line 10. It takes as parameters the pointcut expression of the containing when basic unit and the newly-created stack to create a public pointcut. That pointcut's expression is a conjunction of the basic unit's expression and an *if* pointcut to check whether the top element of the stack is *True* (i.e. if the event got triggered). The next line adds this newly-created pointcut to the containing event.

Testing and Evaluation. In this subsection, we are going to test the simplicity and ease of use of our AspectJ implementation, and evaluate it with respect to our design goals. We will accomplish this by demonstrating how the "named event" pointcut (i.e. the one that is a reference to an event) from the Natural Aspects extension can be implemented in the AspectBench Compiler and we will compare both designs. The steps to implement this functionality in abc are reproduced from similar pointcut extensions described in [21].

1. Extending the lexer

This is the first step when extending abc. In this case, the pointcut we are attempting to implement does not use any new keywords and so no modifications to the lexer are required. Such is the case with our AspectJ implementation as well.

2. Extending the parser

The next step is to extend the parser with a syntax rule for the new pointcut.

In the case of abc the new syntax rule would like this:

```
1.  extend basic pointcut expr ::=
2.    EVENT :x LPAREN formal parameter list opt:a RPAREN : y
3.    {:
4.    RESULT = parser.nf.PCEvents(parser.pos(x,y), a);
5.    :}
```

Listing 8. Extending a Grammar Rule in ABC

The keyword *extend signifies* that the new rule should be added to the rules that already exist for *basic_pointcut_expr.* Line 2 describes the signature (i.e. event reference, followed by a parameter list, enclosed in brackets). Finally, on line 4, the result is a call to the node factory of the parser class.

On the other hand, the same syntax rule written in the CS language of EMFText looks like this:

```
eventdetectors.NamedEventPointcut ::= eventId[] "(" namedEventPattern? ")";
```

Listing 9. Writing a Grammar Rule in Our Implementation

Similarly, this rule also begins with a reference to the event, followed by a pattern enclosed in brackets.

3. Adding new AST nodes

Any functionality with which we wish to extend the abc must be represented in Polyglot's AST. For a clear way of extending the AST, the abc team first suggests to define an interface for the new AST nodes.

```
1.  public interface EventPointcutDecl extends PointcutDecl {
2.    public void registerEventPointcut(EventPointcuts visitor,
3.                                      Context context,
4.                                      EAJNodeFactory nf);
5.  }
```

Listing 10. An Interface for the New AST Node

The next step is to write a concrete class implementing this new interface. Here, some boilerplate code is required (a constructor, methods that allow visitors to visit the pointcut and a concrete implementation of the *registerEventPointcut*). Finally, in order to make sure that we can instantiate this new node, we have to write a subclass to abc's default node factory and create a method for obtaining an instance of the *EventPointcutDecl*.

```
1.  public EventPointcutDecl EventPointcutDecl (Position pos,
2.                                              Event ev,
3.                                              FormalParameterList fpl,
4.                                              String name,
5.                                              TypeNode voidn ) {
6.    return new EventPointcutDecl_c(pos, ev, fpl, name, voidn);
7.  }
```

Listing 11. Obtaining an Instance of the EventPointcutDecl

Now the parser can produce *EventPointcutDecl* when it encounters the appropriate tokens for them.

In the context of our approach, the equivalent process is to extend the language's metamodel, as it is the internal representation of our front-end framework (i.e. EMFText). In our case, this process is more visual than textual, but there are tools (e.g. Emfatic) that offer us the possibility to declare the new

metamodel entity with a textual definition.

```
1.  class NamedEventPointcut extends aspectj.pointcuts.PrimitivePointcut {
2.    val aspectj.patterns.parameterlist.TypeIdWildcardPatternList
    namedEventPattern;
3.    ref commons.Event[1] eventId;
4.  }
```

Listing 12. Adding an Event Pointcut Entity in the Metamodel

4. Adding new join points

The following step is to extend the list of possible join points if necessary. The named event pointcut does not define a new type of join point and so no interaction is necessary here. The same also holds for our AspectJ implementation.

5. Extending the pointcut matcher

Once the corresponding join point shadows have been created in the previous step, writing the back-end files becomes easier. In abc there is a back-end class called *pointcut matcher* that tries every pointcut at every join point shadow found. For the named event pointcut, we have to check whether the current shadow is a *ReferenceShadowMatch* and if so verify that the event being referenced matches the event id that the pointcut syntax rule starts with.

```
1.  protected Residue matchesAt(ShadowMatch sm) {
2.    if (!(sm instanceof ReferenceShadowMatch))
3.      return null;
4.    Reference event ref = ((ReferenceShadowMatch) sm).getReference();
5.    if (!getReferenceId().matchesType(event_ref))
6.      return null;
7.
8.    return AlwaysMatch.v();
9.  }
```

Listing 13. Extending the Pointcut Matcher

This process in abc has no similar equivalent in our approach. The matching of the event from the pointcut to the actual event being referred to in our implementation is done for you by reference resolver classes that EMFText generates automatically. Although sometimes their default behavior is not sufficient and one has to implement his own extensions of them, this was not the case with resolving the named event pointcut.

We are now going to evaluate both designs with respect to the criteria we set out to follow. In terms of simplicity, the abc had to be modified in three out of the five concerns that we introduced. The LOC that were written to solve these three concerns amounted to more than 27. By our plain definition the factor of simplicity is close to a 1/9. In our implementation we had to modify two out of the five concerns for the total amount of 5 LOC, resulting in the superior factor of 2/5.

There are two main reasons behind this success. First, the MDE approach we use offered us an easier solution when we had to extend the abstract representation of the language. Secondly, our approach abstracts over much of the back-end logic (i.e. in this case the pointcut matcher) resulting in less concerns to modify.

In terms of our extensibility criteria, both designs had an similar amount of reused artifacts (i.e. 2 - 3).

5. Conclusion and Future Work

We have presented our implementation of AspectJ and its use as a basis for the development of AspectJ extensions. Our primary design goal of making a foundation that is easier to use due to the disentanglement from the weaving mechanics in AspectJ was met. We demonstrated a case study in which we developed one language extension using our implementation. It served to further evaluate our foundation as modular and extensible. Finally, we conducted a simple comparison between implementing a piece of functionality in both our and an existing open AspectJ implementation (i.e. the AspectBench Compiler). With respect to our criteria for extensibility both implementations fared equally well, but when comparing them by our definition of simplicity the abc was trailing behind. This is due to the fact that our implementation abstracts over much of the hard and complex back-end logic.

There are three directions where our implementation can be improved in future work. First, our implementation does not cover the full 100% AspectJ functionality. We are missing the inter-type declarations, the declare statements of AspectJ and the *percflow* aspect declarations. We left the implementation of these concerns for a future time, although they should not be hard to model. Secondly, there are a few places where the metamodel is "loose" with what is acceptable syntax (e.g. a class can contain an advice block, there can be a proceed statement in any type of advice, not only in an Around). This was done in favor of a more streamlined and easy to follow metamodel. Such invalid cases should be waded out with model validity checks in a post-processing step which we left to implement in the future. Finally, doing more tests on the metamodel and

transformation would give us more confidence in the strength and correctness of our implementation.

References

[1] The AspectJ Team, "The AspectJTM 5 Development Kit Developer's Notebook." [Online]. Available: http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html. [Accessed: 14-Sep-2014].

[2] J. Bézivin, "On the unification power of models," *Softw. Syst. Model.*, vol. 4, no. 2, pp. 171–188, 2005.

[3] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*, Washington, DC, USA, 2007, pp. 37–54.

[4] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[5] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and refinement of textual syntax for models," in *Model Driven Architecture-Foundations and Applications*, 2009, pp. 114–129.

[6] DevBoost, "EMFText User Guide." [Online]. Available: https://github.com/DevBoost/EMFText/blob/master/Core/Doc/org.emftext.doc/pdf/EMFTextGuide.pdf?raw=true. [Accessed: 15-Sep-2014].

[7] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[8] C. Allan, P. Avgustinov, A. S. Christensen, B. Dufour, C. Goard, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, and C. Verbrugge, "Abc the aspectBench Compiler for aspectJ a Workbench for Aspect-oriented Programming Language and Compilers Research," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2005, pp. 88–89.

[9] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for Java," in *Compiler Construction*, 2003, pp. 138–152.

[10] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?," in *Compiler Construction*, 2000, pp. 18–34.

[11] R. Toledo and É. Tanter, "A Lightweight and Extensible AspectJ Implementation.," *J UCS*, vol. 14, no. 21, pp. 3517–3533, 2008.
[12] L. Hendren, O. De Moor, A. S. Christensen, and others, "The abc scanner and parser, including an LALR (1) grammar for AspectJ," 2004.
[13] Eclipse Foundation, "Eclipse Modeling - MMT." [Online]. Available: http://www.eclipse.org/mmt/. [Accessed: 14-Sep-2014].
[14] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *Theory and practice of model transformations*, Springer, 2008, pp. 46–60.
[15] I. Kurtev, "State of the art of QVT: A model transformation language standard," in *Applications of graph transformations with industrial relevance*, Springer, 2008, pp. 377–393.
[16] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1, pp. 31–39, 2008.
[17] "Using black box implementations in Eclipse QVTo | Fábio Levy Siqueira." [Online]. Available: http://www.levysiqueira.com.br/2012/02/black-box-eclipse-qvto/. [Accessed: 15-Sep-2014].
[18] Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, and Richard Paige, "The Epsilon Book." [Online]. Available: http://www.eclipse.org/epsilon/doc/book/. [Accessed: 14-Sep-2014].
[19] The AspectJ Team, "AspectJ 5 Quick Reference." [Online]. Available: http://www.eclipse.org/aspectj/doc/released/quick5.pdf. [Accessed: 15-Sep-2014].
[20] C. Bockisch, S. Malakuti, M. Akşit, and S. Katz, "Making aspects natural: events and composition," in *Proceedings of the tenth international conference on Aspect-oriented software development*, 2011, pp. 285–300.
[21] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "abc: An extensible AspectJ compiler," in *Transactions on Aspect-Oriented Software Development I*, Springer, 2006, pp. 293–334.