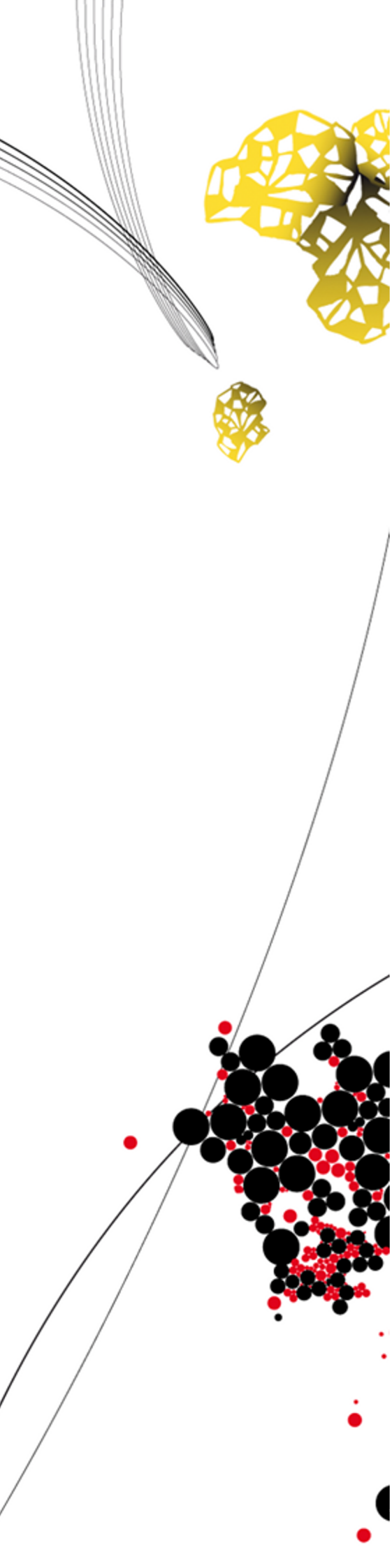

Building an Xtext compiler for Aspect-Oriented languages based on ALIA4J

Göran Blankendal

UNIVERSITY OF TWENTE.

Building an Xtext compiler for Aspect-Oriented Languages based on ALIA4J

M.Sc. Thesis

Göran Blankendal

University of Twente
Department of Electrical Engineering,
Mathematics & Computer Science (EEMCS)
Twente Research & Education on Software Engineering Group (TRESE)
P.O. Box 217
7500 AE Enschede
The Netherlands

Report Date: April 27, 2012
Thesis Committee: Dr. Ing. Christoph Bockisch
Dr. Ivan Kurtev
Ing. Kardelen Hatun

Abstract

A lot of research has been done in the development of aspect-oriented languages and the number of these languages are rapidly increasing. The ALIA4J approach supports the development of these languages by providing an API and runtime model that can be shared among language implementations. Coding the translation to Java code that uses the ALIA4J API is not trivial . This thesis proposes a generic framework for compilers build on Xtext, that performs ALIA4J specific Java code generation such as configuring ALIA4J Attachments. The framework can be re-used for different aspect-oriented language implementations. For the purpose of demonstration, an AspectJ compiler is build that uses this framework. For comparison the compiler is also build without the framework. The compiler that uses the framework turns out to have less bloated code and required less effort from the language developer since the code generation is accomplished by the framework.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Dr. Ing. Christoph Bockish. The completion of this thesis would not be possible without his constant support and guidance. Thanks to Ivan Kurtev and Kardelen Hatun for co-supervising my work.

Thanks to my friends Fred Kaggwa and Tesfahun Aregawy for going through my work and for your comments. I thank my sweet parents for their support and prayers during my studies. Thanks to Tony Santos and Kelvin Kleist for every support given to me. To my classmates Evert Duipmans, Gert Jan and everyone who provided help during my studies I want to say thank you.

My sincere gratitude to my best friend Priscilla Kisubika for her constant encouragement and support. She has always been available in times of need. Above all I want to thank my Lord and Saviour Jesus Christ for giving me the wisdom and ability to complete my studies successfully.

Thank you,
With regards from
Göran Blankendal
April, 2012

Contents

List of Figures

1

Introduction

This thesis proposes to design a generic framework to build compilers for aspect-oriented languages. The framework uses ALIA4J that provides a language-independent meta-model and execution environment. The framework generates Java code that uses the API provided by ALIA4J. This code is an intermediate representation of the advanced-dispatching structure of the program that is compiled. The framework is re-usable among multiple compilers and frees the language developer from writing code generation templates to produce ALIA4J specific Java code for every new compiler that he develops. Reusing this framework results in creating more readable compilers with less effort.

1.1 Current language implementation approach using ALIA4J

A considerable number of advanced-dispatching languages are being developed for research reasons. Advanced-dispatching languages are languages such as aspect-oriented and event-driven languages. It is observed that they share some goals that are realized in different ways. Four of these languages, eAspectJ [8], IIIA [9], EScala [10] and Tracematches [11], have been compared by identifying commonalities and differences. EAspectJ and Tracematches for example both support event composition however eAspectJ realizes this using logical and hierarchical composition but Tracematches uses regular expressions.

To support the commonalities among advanced-dispatching languages, there exists an approach called *ALIA4J* to help building compilers for these languages by offering a language-independent meta-model and execution environment that can be shared among language implementations. ALIA4J also offers meta-model refinements for concrete lan-

guages. The meta-model is implemented as a set of Java classes in the ALIA4J API and is used to express the majority of the semantics of advanced-dispatching language constructs. Using this information, the execution environment derives an execution model of the program's dispatching.

1.2 Problem Statement

Working with the ALIA4J API to express semantics of language constructs and writing a code generation template to produce this code for every new compiler is a tedious task. It leads to lengthy code that is less readable. It is therefore important for the language developer to have a layer of abstraction that will free him from working with the ALIA4J API directly and writing code generation templates for this. The language developer needs a framework that will enable him to develop compilers for different aspect-oriented languages using ALIA4J with minimal effort.

1.3 Approach and Contribution

This thesis proposes to design a re-usable and generic Xtext-based framework for aspect-oriented language compilers, that performs ALIA4J specific code generation. Xtext is a framework that can be used to develop domain-specific or general purpose languages. It includes the ability to generate a parser for a language described by an Xtext grammar, an abstract syntax tree (AST), scoping components and a code formatter. In addition it generates an eclipse-based IDE that is tailored for the language providing code highlighting and content assist. Xtext uses the concept of Ecore (meta-)models internally for activities such as validating the program structure and generating code for it. When a program is compiled, a model is created that represents the program's structure. From this model, another model is created that represents the structure of the Java code that will be generated.

The proposed framework contains a meta-model for ALIA4J in the form that can be used with Xtext. It allows both an ALIA4J model and a Java model to be inferred from the model that represents a parsed program. The Java model contains that Java part for the program. AspectJ for example allows Java code in the advice body of an aspect. The ALIA4J model however contains dispatch related information of the program. It is used to generate appropriate Java code for configuring ALIA4J Attachments and Specializations using the ALIA4J API.

The framework is used to realize a compiler for the aspect-oriented language AspectJ. This implementation turns out to be more readable and concise then an implementation without the framework. It also demonstrates the extensibility of the framework with regard to adding refined meta-model entities.

1.4 Outline

The remainder of this thesis is structured as follows.

Chapter 2 gives a discussion on the motivation of the work presented in this thesis.

Chapter 3 presents the necessary technical background information to understand the framework.

Chapter 4 gives the architecture of the framework in detail.

Chapter 5 describes an implementation of a compiler for the AspectJ language and gives an evaluation of the framework.

Chapter 6 summarizes this thesis and presents future work.

2

Motivation

2.1 Different languages sharing the same concepts

A lot of research has been done in the development of advanced-dispatching languages and the number of these languages are rapidly increasing.

Dispatching is the execution mechanism that resolves abstractions and binds concrete functionality to their usage. A common example of dispatching in object-oriented languages is receiver-type polymorphism: Whenever a method is invoked, the runtime environment chooses between different implementations of the method in the type hierarchy depending on the dynamic type of the receiver object. The term *dispatch site* is used to refer to the place from where the method is called or a field accessed. Languages that go beyond this traditional received-type polymorphism are referred to as *advanced-dispatching languages*. In these cases a dispatch can consider additional and more complex runtime states, and that functionality can be composed in different ways. Languages that are event-driven or aspect-oriented are advanced-dispatching languages.

We have observed that different event-driven and aspect-oriented languages share some goals but realize them partially in different ways. In particular a comparison has been done for the languages eAspectJ [8], IIIA [9], EScala [10] and Tracematches [11]. Commonalities have been identified by investigating mechanisms that languages use to support their features. The following section 2.1.1 gives an explanation on event-driven and aspect-oriented programming. Section 2.1.2 then defines the comparison criteria and compares those languages using this criteria.

2.1.1 Event-driven and aspect-oriented programming

The event-driven programming paradigm in object-oriented languages is based on *imperatively triggered events*. By imperative we mean that there is some user-defined code that realizes the triggering of an event. Triggering an event means to announce to the running program that an event has occurred. A part of the program that depends on the occurrence of the event is notified and reacts to that event by executing code. This so-called *subscriber* can react to multiple events by registering to more then one event type. It is also possible to combine events into more complex events by forming a hierarchy: A subscriber can react to an event and trigger a new event. Registering a subscriber can only be done in terms of an event type. When the developer is only interested in events that have specific properties (e.g. method arguments), a subscriber still needs to be registered to an event type. In order to select a more specific occurrence of the event, the event properties need to be checked imperatively before reacting to the event.
The aspect-oriented programming (AOP) paradigm features *implicit* events. Implicit events (e.g. method calls) are announced by the execution environment. In AOP these are called join points. A pointcut is a construct that selects join points and thus can be viewed as a declarative event description or selector. As opposed to the event-driven paradigm, a pointcut can additionally refer to properties of the event such as method names and arguments. If constraints on these properties are satisfied, the event is selected. The equivalent of a subscriber in AOP is an advice. This is combined with a pointcut within an aspect.
In both paradigms, an event is created and a subscriber can react to the event. There are notable differences however. In the event-driven programming paradigm the focus is on *triggering the event at the right place*. In the AOP paradigm we focus on *selecting the right event instances*.
In the following section a comparison is made between languages that have characteristics of both paradigms. The term *event description* is used to refer to the mechanism to select an event instance.

2.1.2 Comparing the languages

The criteria that is used to compare those languages are described in the following. A visualization of the comparison between the investigated languages is depicted in the table below.

History-based information There are various reasons why we may want to access the history of a program. One reason is to access past information to influence current decisions. We examined how access to past events or data can be expressed. Do the languages support *imperative code* to collect this information or do they support *declarative definitions*? It turned out that all the investigated languages support history-based information. However the approaches have different ways of computing history-based information and executing advice code based on this.
Tracematches support declarative retrieval of history while EScala has no support for this at all. eAspectJ and IIIA support declarative history only through the `cflow` and `cflowbelow` constructs inherited from AspectJ.
Both eAspectJ, IIIA and EScala support imperative code to collect history.

Event Description and Abstraction Another criteria is the re-usability of event descriptions and the way they can be composed into more complex descriptions that are capable of selecting multiple events. How do different approaches allow the composition of event descriptions to be expressed? This composition may be supported by different mechanisms like using special operators, inheritance, etc.
Event composition by binary operators is supported in all the investigated languages except Tracematches. In Tracematches events are composed in the form of regular expressions over joinpoints. In eAspectJ it is besides binary composition also possible to declare hierarchies of events by allowing an event to trigger a more abstract event.

Description	eAspectJ	IIIA	EScala	Tracematches
declarative history	`cflow`, `cflowbelow`	`cflow`, `cflowbelow`	No	traces
imperative history	Yes	Yes	Yes	No
event composition	logical + hierarchical	logical	logical	RE
re-usability	events + subscribers	events + subscribers	events + subscribers	tracematch

2.2 Implementing advanced-dispatching languages

As stated in the previous subsection, different languages share the same concepts however they are implemented in different ways. Both IIIA and Tracematches are realized

differently using the AspectBench Compiler (abc) [12]. EScala is not implemented using abc. eAspectJ does not have a compiler implementation currently.

Compilers translate from high-level language to low-level language. A traditional compiler consist of several phases such as the following:

Syntactical analysis: The source program is parsed and checked whether it conforms to the syntax of the source language. In this phase an *Abstract Syntax Tree* (AST) is build to represent the program structure. This representation is then used in the following phases.

Context analysis: The parsed program is analysed to check whether it conforms to source language contextual constraints. An example is to check if the type of a variable corresponds to the value that an expression assigns to it.

Code generation: Low-level language code is generated according to the semantics of the source language.

When developing a compiler, one should pay attention to all those phases. The code generation phase is more complex for advanced-dispatching languages. Aspect-oriented languages such as AspectJ weave code with some other program code to obtain the final result.

New languages are often developed as extensions of existing languages where the source code of the extension is transformed to the intermediate representation of the language that is extended. Compiler frameworks allow code transformations to be re-used for languages that are related in syntax. An example is the AspectBench Compiler (abc) that is specifically designed for implementing extensions to AspectJ. The frameworks do not allow this re-use across language families. The next section therefore outlines an approach that makes this possible.

2.3 The ALIA4J solution to shared concepts in different languages

As described in the previous subsection, compiler frameworks do not allow code transformations to be re-used across language families. Developing compilers for languages that share the same concepts such as the ability to gather historic information, leads to redundant work. The goal of ALIA4J [1] is to ease the burden of implementing advanced-dispatching languages. ALIA4J adds a layer of abstraction by providing a

language-independent meta-model of dispatching as an intermediate language. It also provides an execution environment that can be reused among different language implementations.

2.3.1 The Language Independent Advanced-dispatching Meta-model

Dispatch-related constructs of advanced-dispatching languages largely overlap in their semantics and the majority of the semantics of these constructs can be described in a language-independent way using the meta-model provided by ALIA4J. This meta-model of advanced-dispatching declarations is called LIAM [1]. LIAM is implemented as a set of plain Java classes. An abstract view of the meta-model is shown in Figure 2.1. Core concepts of various dispatching mechanisms are captured in LIAM as an intermediate language.

The representations defined in this intermediate language are used to automatically derive an execution model of the program's dispatching. This is done using a framework for execution environments, called FIAL [2], that is additionally provided by ALIA4J.

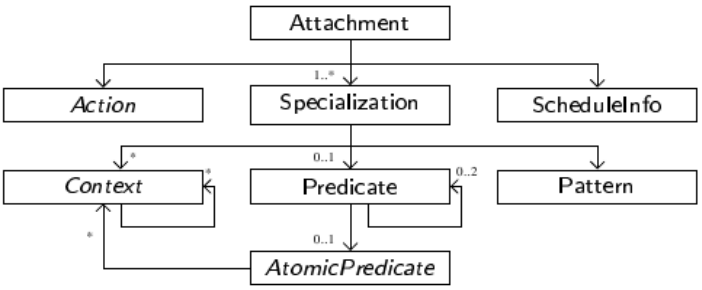

Figure 2.1: Abstract view of the Language Independent Advanced-dispatching Meta-model

The abstract meta-model is refined for concrete languages. There are for example different implementations for the LIAM Action, Context, Predicate and Pattern. The entities and refinements of this meta-model are explained in the following paragraphs.

Pattern. A LIAM Pattern describes a dispatch site. There are five predefined subclasses of Pattern offered by LIAM to model types of dispatch sites such as calls to a method, constructor or static initializer, and reading or writing fields.

[1] The Language Independent Advanced-dispatching Meta-model. See http://www.alia4j.org/alia4j-liam/

[2] The Framework for Implementing Advanced-dispatching Languages. See http://www.alia4j.org/alia4j-fial/

Context. A Context models the dependency on runtime values. Contexts are able to expose different kind of values available during dispatch such as, e.g., the `ArgumentContext` that captures a single argument value and the `CallerContext` to capture the receiver object.

Atomic Predicate. An `AtomicPredicate` provides the ability to test values that are available during dispatch. For example a dynamic type check of an argument can be accomplished using an `InstanceOfPredicate` parameterized with an `ArgumentContext`.

Predicate. Composite predicates are provided to allow a more complex selection of a joinpoint. This is possible by allowing the construction of trees with inner nodes as conjunctions (`AndPredicate`) or disjunctions (`OrPredicate`) and leafs (`LeafPredicate`)

Specialization. A Specialization entity associates a Pattern with a Predicate to select specific calls. It also allows the declaration of a list of contexts to define which runtime values need to be exposed to Actions at the selected dispatch.

Action. After the evaluation of predicates, an action needs to be performed. This is modelled using a LIAM Action entity such as a `MethodCallAction` to call an individual method or a `ThrowAction` to raise an exception.

Attachment. A Specialization is associated with an Action using an Attachment entity. When a dispatch is selected using the Pattern and Predicate in the Specialization, the context values defined by the Specialization are passed as arguments to the Action.

Schedule Information The Schedule Information that is associated with an Attachment, specifies when the Action is executed relative to the dispatch site. This can be `Before`, `After` or `Around` similar to AspectJ.

2.3.2 Translation to ALIA4J Entities

In this section, the compilation from an advanced-dispatching language to the ALIA4J intermediate language is explained. This compilation is not trivial since the output should be ALIA4J specific API calls in Java. This leads to a lot of accidental complexity. As explained in section 2.2, a compiler consists of several phases. This section however focusses on the code generation phase. Using an example program written in the language AspectJ, it is shown how the output of ALIA4J source code should look like. Since the

focus in this section is not on how to build a compiler, we assume that the language developer creates this output source code manually.

Consider the program shown in Listing 2.1. This program defines a pointcut, named *secureAccess*, that selects an execution of the method named *update* from the class *model.Account*. There is an advice block declared that will run before the method is executed. This advice code consists of a statement that prints the message "*Security Applied!*" to the console.

Listing 2.1: Small AspectJ program

```
1 public aspect Security {
2   pointcut secureAccess() : execution( * model.Account.update(..));
3   before() : secureAccess() {
4       System.out.println("Security Applied!");
5   }
6 }
```

When investigating the code produced by the AspectJ compiler, we notice that a class is created for each defined aspect. All the standard Java members such as fields, methods and static initializer defined in the aspect are contained in this class. It also contains a virtual method for each advice.

Using the same approach, a class *Security* is derived from the aspect by the language developer with a method named `adviceMethod` to contain the advice code (Listing 2.2).

Listing 2.2: A Java class containing the advice code

```
public class Security {
  public void adviceMethod () {
      System.out.println("Security Applied!");
  }
}
```

After this, the semantics of the AspectJ constructs need to be expressed using ALIA4J. Listing 2.3 shows that this is done using ALIA4J API calls in the body of the method `performImport` of a class that implements the `org.alia4j.fial.Importer` interface. This method is executed before the program starts and is used to programatically deploy the ALIA4J Attachments.

Pattern. LIAM comes with at least four refinements of Pattern: *ConstructorPattern*, *FieldPattern*, *MethodPattern* and *StaticInitializerPattern*.

- The ConstructorPattern is used to match a constructor call. This corresponds to `.new(..)` in AspectJ.
- The FieldPattern is used to match read or write access to a field. The `get(..)` and `set(..)` are used respectively in AspectJ.
- The MethodPattern is used to match the execution of a method. This corresponds to `call(..)` and `execution(..)` in AspectJ.
- The StaticInitializerPattern is used to match the static initializer of a class. In AspectJ this is accomplished using the `staticinitialization(..)` construct.

Since the pointcut in this AspectJ program selects a method execution, a `MethodPattern` is used to capture this information in Line 6-12. This Pattern specifies the selection of a method with the exact name `update` from the exact type `model.Account`. It allows any modifier of the method, return type, number of parameters and any exceptions to be declared in the method signature.

Listing 2.3: Mapping the AspectJ program to ALIA4J

```
1  public class Importer implements org.alia4j.fial.Importer {
2
3    @Override
4    public void performImport() {
5
6        MethodPattern pattern = new MethodPattern(
7        ModifiersPattern.ANY,
8        TypePattern.ANY,
9        new ExactClassTypePattern(TypeHierarchyProvider.
             findOrCreateFromNormalTypeName("model.Account")),
10       new ExactNamePattern("update"),
11       ParametersPattern.ANY,
12       ExceptionsPattern.ANY);
13
14       Predicate<AtomicPredicate> predicate = TruePredicate.
15         <AtomicPredicate> truePredicate();
16
17       Specialization specialization = new Specialization(pattern,predicate
             ,
18         Collections.<Context>singletonList(
19           ContextFactory.findOrCreateLazyObjectConstantContext(
20             TypeHierarchyProvider.findOrCreateFromNormalTypeName("security
                   .Security")
21             )
22           )
23       );
24
```

```
25      Action action = ActionFactory.findOrCreateMethodCallAction(
26            TypeHierarchyProvider.findOrCreateFromNormalTypeName(
27              "security.Security"),
28            "adviceMethod",
29            TypeHierarchyProvider.findOrCreateFromClasses(new Class[] {  }),
30            TypeHierarchyProvider.findOrCreateFromClass(Void.class),
31            ResolutionStrategy.VIRTUAL);
32
33        Attachment attachment = new Attachment(Collections.singleton(
              specialization), action, ScheduleInfo.BEFORE);
34
35        org.alia4j.fial.System.deploy(attachment);
36    }
37 }
```

Predicate. LIAM comes with at least five refinements of Predicate: *OrPredicate*, *AndPredicate*, *BasicPredicate*, *FalsePredicate* and *TruePredicate*.

- The OrPredicate or AndPredicate are used to respectively calculate the logical disjunction or conjuction of two predicates.
- The BasicPredicate is parameterized by an AtomicPredicate and can be used to negate the value returned by the AtomicPredicate.
- The FalsePredicate and TruePredicate are used to return either a false or true value respectively.

LIAM comes with at least four refinements of AtomicPredicate: *InstanceofPredicate*, *ExactTypePredicate*, *MethodPredicate* and *ThresholdPredicate*.

- An InstanceofPredicate is used to dynamically check the type of any object provided by a Context. The predicate is parametrized with the Context and the required type of the argument. An example where this is used is for the `target`, `this` or `args` pointcut designators in AspectJ. In the case of `args`, the InstanceofPredicate is used in combination with an ArgumentContext to check the type of an argument.
- The ExactTypePredicate is used in the same way as the InstanceofPredicate. The only difference is that the type of the object must be exactly the same as the required type.
- A MethodPredicate is evaluated by invoking a method with a boolean return type. This is used to realize the `if` pointcut designator in AspectJ.

- The ThresholdPredicate is used to calculate whether a value exceeds a pre-defined threshold.

Since the AspectJ pointcut does not contain any construct based on some dynamic value of the joinpoint context such as the `this` or `args` keyword, we use a predicate that always evaluates to *true* (Line 14-15).

Specialization. Both the Pattern and the Predicate are combined in a Specialization (Line 17-23).

Context. There are at least five refinements of Context: *ArgumentContext*, *CalleeContext*, *CallerContext*, *ConstantObjectContext* and *LazyObjectConstantContext*.

- The ArgumentContext captures a single argument value. This is used for the `args` pointcut designator in AspectJ.
- The CalleeContext captures the callee of an execution. The `target` pointcut designator in AspectJ is an example where this is used.
- The CallerContext captures the callee of an execution. The `this` pointcut designator in AspectJ is an example where this is used.
- The ConstantObjectContext exposes a specific instance.
- The LazyObjectConstantContext creates an instance of a specified class upon first use and exposes the same instance at subsequent uses.

The Specialization in line 17 takes a list of context values. The first context value in ALIA4J should correspond to the instance of the aspect. This context is necessary because the instantiation strategy of AspectJ requires that an instance of the aspect be created. Since the example program does not specify a *per-clause* in the aspect-header, the *issingleton* is used as default. This AspectJ construct specifies that the same instance must be used for each advice execution. This instance must be created the first time an advice of the aspect is executed. The `LazyObjectConstantContext` parameterized with the aspect class `security.Security` is used to achieve this result (Line 19-22).

Action. LIAM comes with at least three refinements of Action: *MethodCallAction*, *FieldReadAction* and *FieldWriteAction*.

- The MethodCallAction is used to invoke a method.

- The FieldReadAction and FieldWriteAction are used to execute a field read or write action.

Since there is a `execution` pointcut designator used in the AspectJ program, a `MethodCallAction` is created (Line 25-31) to specify that the method containing the advice code will be called when the dispatch is selected and the predicate evaluated. This action specifies the name of the class "`security.Security`" containing the advice code , the name of the method "`adviceMethod`", a list of the parameter types which in this case is empty and the return type which is `void`. It also defines the ResolutionStrategy which specifies that the method is called on an instance (`ResolutionStrategy.VIRTUAL`). If the method is static than the `ResolutionStrategy.STATIC` should be used.

Attachment. An Attachment associates the Specialization and the Action to be performed (line 33). The `ScheduleInfo.BEFORE` specifies that the Action will be performed before the dispatch site.

Deployment. At last the Attachment is deployed to the execution environment.

2.3.3 Problem statement

As can be seen from Listing 2.3, the translation to ALIA4J Java code is not trivial. Writing code generation templates to produce this Java code is as a result much more complex. The example code in Listing A.1 of Appendix A shows how complex and bloated a template can be. This thesis proposes a generic framework to build compilers for aspect-oriented languages that will hide ALIA4J specific code generation from the language developer. The goals and requirements for this framework are described in the following section.

2.3.4 Goals and Requirements

The goals and requirements for the framework are explained in the following paragraphs.

Re-usability. Although there exists a meta-model and an execution environment that can be used by multiple languages, there is still redundant work involved when building multiple compilers to ALIA4J. This redundancy is mainly in the translation to ALIA4J specific Java code. Coding this translation has to be done in all those compilers and thus is a redundant activity. From section 2.3.2 we observe that the translation from an advanced-dispatching language to the ALIA4J Java code is not trivial. Writing a code

generator for this translation increases the complexity.
Even when using the Xtext framework as described in section 3.1 to build multiple compilers to ALIA4J, the language developer is still involved in writing an ALIA4J specific code generation for every new compiler. Ideally the language developer should not be required to do this, and should only focus on LIAM and the high level language constructs. We want to hide the complexity of the ALIA4J code generation in Xtext in a reusable component.
The framework will have the task of generating Java code specific to ALIA4J which will free the developer from this tedious task. The framework should be reusable among different aspect-oriented compilers. This should lead to faster development of aspect-oriented language compilers.

Readability. Writing templates for code generators is not easy. Templates are complex because they are meta-programs that can produce code for multiple programs. Template programs are therefore in general more difficult to understand.

When building compilers using ALIA4J, templates need to be developed by the language developer to produce code such as the translated ALIA4J Java code shown in Listing 2.3. This code is six times longer then the AspectJ program from section 2.3.2. This is because factory methods are used to create concrete instances of LIAM entities such as Contexts and Actions in Line 19 and 25 of Listing 2.3. The ALIA4J API also uses naming conventions for these methods. This introduces an amount of inconvenience and leads to bloated code which is less readable. The template to produce this code is as a result much more bloated. See as an example lines 3-17 of the code in Listing A.1 of Appendix A. The mix of ALIA4J Java code with conditions and loops to generate the right code shows how complex and bloated a template can be.

We therefore want to hide this template from the language developer and provide this as a service. We intend to develop a re-usable code generator containing this template that will ultimately improve the overall readability of a compiler implementation.

Extensibility. As discussed in section 2.3.1, ALIA4J provides an abstract language-independent meta-model with refinements for concrete languages. When implementing a new language following the ALIA4J approach, it might be the case that the semantics of the language cannot be expressed using existing meta-model refinements. In this case new LIAM refinements need to be added. This effects the code generation component since new code should be generated for the new entity. Ideally the framework should not have to be changed. The framework should be extensible by supporting the addition of

LIAM refinements for new languages.

Complexity is a possible goal that is not targeted in this thesis. Generating ALIA4J code can become very complex. Consider as an example the following AspectJ pointcut expression:

```
(( call(x)|| get(y))&& this(F1))|| set(z)&& !(target(F2)&& args(..,F3)).
```

The `call(x)`, `get(y)` and `set(z)` constructs will be mapped to three different ALIA4J Specializations since only one pattern is allowed per Specialization. The `this(F1)`, `target(F2)` and `args(..,F3)` will be mapped to three ALIA4J predicates. However ALIA4J only allows the negated normal form of a predicate. This makes it even more complex since this expression first has to be transformed into the negated normal form. This complexity is also shared among compilers since multiple languages allow conjunctions and disjunctions in their pointcut expressions. This kind of complexity could be hidden from the language developer by allowing the framework to do this transformation.

3

Background

3.1 Xtext

Since the generic compiler framework is build on top of Xtext, this section gives the necessary background information to understand the framework. Xtext [2] is a framework that is used to develop domain-specific or general purpose languages. It includes the ability to generate an eclipse-based IDE that is tailored for the language providing syntax highlighting and content assist.

Assume that a language developer wants to build a compiler based on Xtext for a language called *Alpha*. The language developer first writes a grammar for this language and based on this grammar file, Xtext will generate a lexer, parser and a meta-model for the language.

Xtext also generates a skeleton for a class called *AlphaJvmModelInferrer*. This class is written in a language called *Xtend* and will be compiled to Java. This is further described in section 3.3. The AlphaJvmModelInferrer is used to map the concepts from the Alpha meta-model to concepts of the Jvm meta-model. This is a meta-model that comes with Xtext and describes the AST of the Java language. A *JvmModelGenerator* is used to generate Java code from this Jvm model.

When running the compiler, the program text will be parsed and a model will be created based on the language meta-model. This model will be transformed into a Jvm model from which Java code will be generated. This Java code is then compiled to byte code using the Java compiler.

Language models. Since Xtext relies on the Eclipse Modeling Framework (EMF) [7] internally, it generates a meta-model that describes the *Abstract Syntax Tree* (AST) of

the language Alpha. This AST is an abstraction of the syntactical information of the program. The language in which the meta-model is defined is called *Ecore*. Ecore is an important part of EMF. EMF models are used by Xtext as the in-memory representation of any parsed program.

Figure 3.1, depicts the generation of a model during runtime. The model contains relevant information from the parse tree. We refer to this model as the *language model*.

Figure 3.1: A Parser parses a program and a model is created

Model Inference. The *Model Inference* part of Figure 3.3 depicts the transformation by the `AlphaJvmModelInferrer` of a language model Alpha to a so-called *Jvm model*. This is a model that conforms to the Jvm meta-model. Part of this meta-model is shown in Figure 3.2. The classes defined in the meta-model that are mainly referred to in this report are listed below.

- A `JvmOperation` corresponds to a Java method.
- A `JvmField` corresponds to a Java field.
- A `JvmGenericType` corresponds to a Java class or interface. This class contains a property called *members* to refer to class members such as Java fields and methods.

The process of transforming one model to another model in Xtext is called *inferring*. The skeleton class `AlphaJvmModelInferrer` needs to be implemented by the developer so that a correct Jvm model is inferred. The output of this inferring process is an instance of the root of the Jvm meta-model, the `JvmGenericType`. To make the code of the `AlphaJvmModelInferrer` readable and concise, the developer can make use of the `JvmTypesBuilder`. This is a helper class that provides a lot of extension methods used to generate the Jvm model. See section 3.3 for an explanation of extension methods.

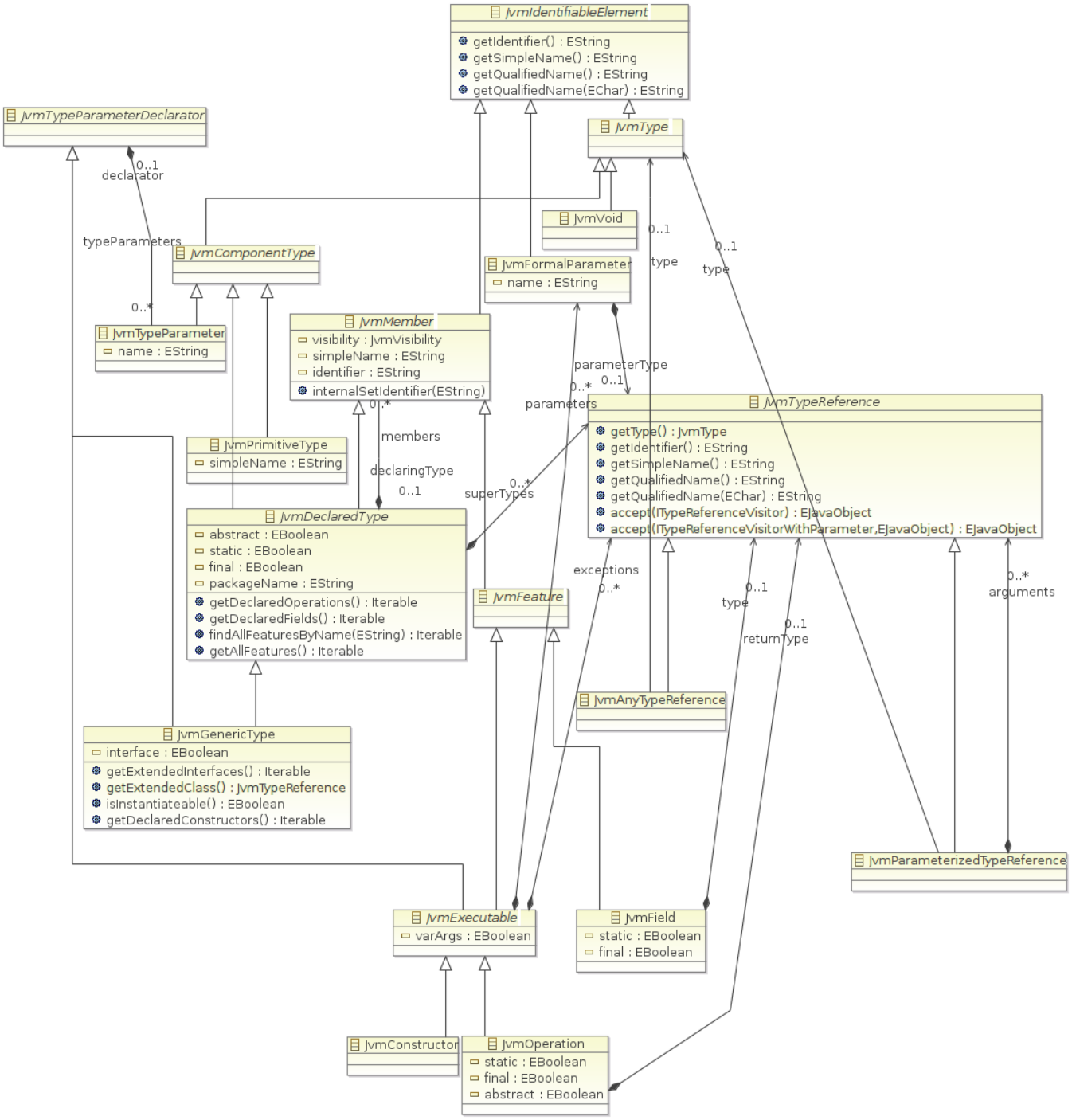

Figure 3.2: Part of the Jvm meta-model that comes with Xtext

Code Generation. The `JvmModelGenerator` which is an implementation of the `IGenerator` interface, is provided by Xtext to generate Java code from a Jvm model such as Java classes and methods. It has as input a resource object with a collection containing both the language model and the Jvm model and can output multiple files to the file system. This process is shown in the *Code Generation* part of Figure 3.3.

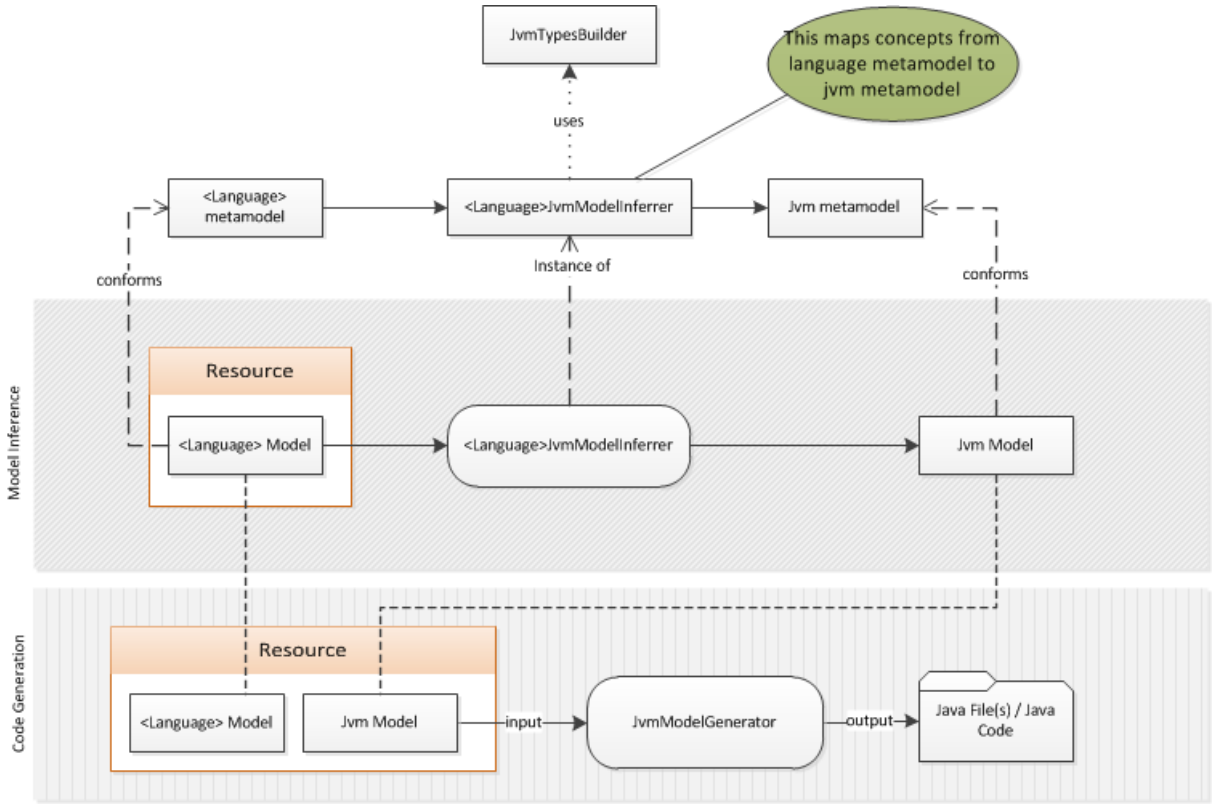

Figure 3.3: Simplified Xtext framework

3.2 Dependency Injection with Google Guice

One important facet when using Xtext is that it uses Dependency Injection (DI) as defined by JSR-330 [5]. A Java application typically consists of objects that collaborate with each other. This means that objects depend on other objects to perform a particular task. Dependency Injection allows a framework to provide an instance of a class that is depended on, instead of initializing this class by itself. The Xtext framework heavily relies on DI to configure components within the framework. This minimizes the coupling between components and enables the developer to use some other component

implementation with minimal effort.

Google Guice [6] is the reference implementation of JSR-330 and is used by Xtext to provide the DI features. Suppose that a class wants to use an `IJvmModelInferrer`. DI can be achieved by declaring a field and annotating this using the `Inject` annotation as Listing 3.1 shows.

Listing 3.1: Using DI in the JvmModelAssociator

```
1 @Singleton
2 public class JvmModelAssociator implements IJvmModelAssociator,.. {
3
4   @Inject
5   private IJvmModelInferrer inferrer;
6   ...
7 }
```

When the `JvmModelAssociator` class is instantiated, Guice sees that it requires an instance of `IJvmModelInferrer`. In Xtext many instances are created by Guice itself. The way for Guice to know how to instantiate classes for declared dependencies is by using a *Module*. An abstract type is mapped to a concrete class using this module. Xtext uses an enhanced version of Guice's Module API where certain methods are reflectively invoked to find which class to instantiate for a declared dependency. See as an example Listing 3.2.

Listing 3.2: A Module defining a binding with a concrete type

```
1 public abstract class AbstractAspectJRuntimeModule extends
      DefaultRuntimeModule {
2   ...
3   public Class<? extends org.eclipse.xtext.xbase.jvmmodel.
        IJvmModelInferrer> bindIJvmModelInferrer() {
4     return org.xtext.java.aspectj.jvmmodel.AspectJJvmModelInferrer.class;
5   }
6 }
```

In this example a binding is declared between the `IJvmModelInferrer` and the `AspectJJvmModelInferrer`. The method name is the text "*bind*" followed by the name of the abstract type which in this case is `IJvmModelInferrer`. The method returns a concrete type, in this case `AspectJJvmModelInferrer`. This means that whenever Guice finds that an instance of `IJvmModelInferrer` is required, an instance of `AspectJJvmModelInferrer` will be assigned to the specified field. The instantiation and sharing can be configured to specify whether a new instance is returned or one instance is shared across the application.

3.3 Xtend

Xtend [4] is a programming language that is used in addition to Java when working with Xtext. The Xtend Reference Documentation [4] gives the following definition of Xtend:

> "Xtend is a statically-typed programming language which is tightly integrated with and runs on the Java Virtual Machine. It has its roots in the Java programming language but improves on many concepts.."

The concepts referred to are among others: Xtend translates to Java, the use of extension methods and closures. The code shown in Listing 3.3 is taken from the Xtext documentation [3] and demonstrates some of these features.

Listing 3.3: Snippet of Xtend code of DomainModelJvmModelInferrer [3]

```
1
2  @Inject extension JvmTypesBuilder
3
4   def dispatch void infer(Entity element,
5                  IAcceptor<JvmDeclaredType> acceptor,
6                  boolean isPrelinkingPhase) {
7
8     acceptor.accept(element.toClass(element.fullyQualifiedName) [
9       documentation = element.documentation
10      for (feature : element.features) {
11        switch feature {
12          Property : {
13            members += feature.toField(feature.name, feature.type)
14            members += feature.toSetter(feature.name, feature.type)
15            members += feature.toGetter(feature.name, feature.type)
16          }
17          Operation : {
18            members += feature.toMethod(feature.name, feature.type) [
19              for (p : feature.params) {
20                parameters += p.toParameter(p.name, p.parameterType)
21              }
22              documentation = feature.documentation
23              body = feature.body
24            ]
25          }
26        }
27      }
28    ])
29  }
```

Translates to Java. Xtend code is translated to Java code instead of byte code. This means that the Java code can be viewed to understand what is going on as well as

debugged when writing code in Xtend.

Extension methods. Extension methods enhance closed types with new functionality optionally injected via JSR-330 [5]. Line 2 of Listing 3.3 shows how an instance of `JvmTypesBuilder` is injected in the model inferrer. This `JvmTypesBuilder` defines methods that act as an extension in the model inferrer.

Take as an example the `element` parameter of type `Entity` in line 4. This type is defined in the language meta-model. A `toClass` method is invoked on an object of this type in line 8 with the expression `element.toClass`. This method however is not defined in the type `Entity` but in the `JvmTypesBuilder`. Listing 3.4 shows an example of some methods defined in the `JvmTypesBuilder` that are used as an extension in the model inferrer.

Listing 3.4: An example of methods defined in the JvmTypesBuilder

```
1  public JvmGenericType toClass(EObject sourceElement, QualifiedName name,
       Procedure1<JvmGenericType> initializer) {
2    return toClass(sourceElement, name!=null?name.toString():null,
       initializer);
3  }
4  public JvmGenericType toClass(EObject sourceElement, String name,
     Procedure1<JvmGenericType> initializer) {
5    final JvmGenericType result = createJvmGenericType(sourceElement, name
       );
6    if (result == null)
7      return null;
8    if(initializer != null)
9      initializer.apply(result);
10   ...
11 }
12 public JvmOperation toMethod(EObject sourceElement, String name,
     JvmTypeReference returnType,
13     Procedure1<JvmOperation> init) {
14   JvmOperation result = TypesFactory.eINSTANCE.createJvmOperation();
15   result.setSimpleName(nullSaveName(name));
16   result.setVisibility(JvmVisibility.PUBLIC);
17   result.setReturnType(cloneWithProxies(returnType));
18   if (init != null && name != null)
19     init.apply(result);
20   return associate(sourceElement, result);
21 }
```

The code `element.toClass(element.fullyQualifiedName)[..]` of Listing 3.3 line 8 translates into a call to the `JvmTypesBuilder` as can be seen in Listing 3.5 line 4. The `element` variable on which `toClass` is invoked, becomes the first parameter of the method

call in Java.

Listing 3.5: Example Java code translated from Xtend

```
1    final Procedure1<JvmGenericType> _function = new Procedure1<
         JvmGenericType>() {
2      public void apply(final JvmGenericType it) {..}
3    };
4  JvmGenericType _class = this._jvmTypesBuilder.toClass(element, name,
       _function);
5  ...
```

The use of extension methods makes the code in the model inferrer more concise because the `JvmTypesBuilder` can take care of instantiating and initializing Jvm types instead of the inferrer. This leads less and concise code.

Closures. Closures are expressions which produce anonymous functions. A closure can be an argument of an extension method by appending it to the method call. For illustration see the code in square brackets between line 8 and 28 of Listing 3.3. This code produces a function object as can be seen in Listing 3.5 line 1. This function object is then passed to the `toClass` method of the `JvmTypesBuilder`. The function object is applied on the resulting `JvmGenericType` using `initializer.apply(result)` in line 9 of Listing 3.4. This means that the expressions in the closure are executed in the context of the `JvmGenericType`. See as an example the expression `members += feature.toField(feature.name, feature.type)` of line 13 in Listing 3.3. This results in creating a `JvmField` and adding this to the `members` list attribute of the `JvmGenericType`.

4

The compiler framework

This section describes the compiler framework built upon Xtext. The compiler parses a language model from a program text file. From the language model both a Jvm model and an ALIA4J model is inferred. Another Jvm model is inferred from the ALIA4J model to contain the ALIA4J API calls to setup ALIA4J Attachments in Java. The Jvm and ALIA4J models are used to generate code in textual form.

4.1 Reusable ALIA4J framework

Section 2.3.1 explained that ALIA4J provides a meta-model called LIAM that is implemented as a set of Java classes. For this meta-model a grammar can be defined to describe the syntax of a hypothetical language called *ALIA4J*. XText would infer an Ecore meta-model from this which ideally resembles LIAM.

Instead of defining a grammar, this thesis proposes an Ecore version of LIAM. This section explains this Ecore meta-model. It also explains part of the framework that infers a Jvm model from an ALIA4J model. The rest of the framework is explained in section 4.2. This inferring process of a Jvm model from an ALIA4J model is depicted in Figure 4.1. The components with a background color are key contributions to the generic part of the framework.

Language models. To make LIAM usable in Xtext, we created an Ecore version of this meta-model. Part of this is depicted in Figure 4.2 and 4.3. When the Xtext-based compiler is executed to compile a program written in the language *ALIA4J*, a model will be generated based on the ALIA4J meta-model. This model corresponds to pointcut and advice configuration for an aspect-oriented language.

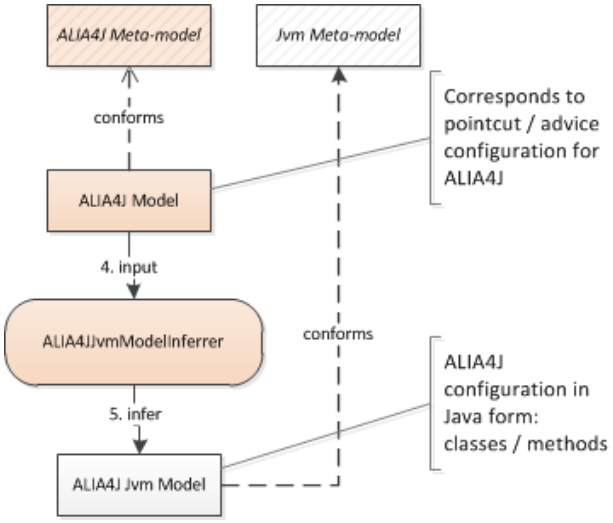

Figure 4.1: Inferring a Jvm model from an ALIA4J model

The meta-model defines classes such as the `ALIA4JAttachment` and `ALIA4JSpecialization`. Different from LIAM is the fact that this Ecore version contains an additional `ALIA4JAspect` that corresponds to an aspect configuration. It contains a collection of Attachments of which each corresponds to an advice/pointcut combination. It also contains an `aspectFQN` String property to hold the fully qualified name of the aspect. Languages that are not aspect-oriented may not have such a construct. In this case, the language developer can assign any String to the `aspectFQN` property of the `ALIA4JAspect`.

Additionally, an Attachment in the Ecore version can have one instead of multiple Specializations. This however is a simplified version and is not an inherit limitation.

A reference between two entities in Ecore can be modeled using a *containment* or a *non-containment* reference. As opposed to the latter, a *containment* reference from entity A to entity B specifies that entity B is part of entity A and cannot exist by itself. Ecore requires all entities to be contained by a resource directly or indirectly. An entity is indirectly contained by a resource when it is contained by another object that is contained by a resource. Because of this requirement an `ALIA4JAttachment` references a `ALIA4JSpecialization` using a containment reference. An `ALIA4JAttachment` is in turn contained by an `ALIA4JAspect` which is contained by a resource. This is important to remember for the discussion later in this section.

Another difference is that the `ALIA4JSpecialization` contains a name property of a String type. This name is used during the code generation that is described later in this

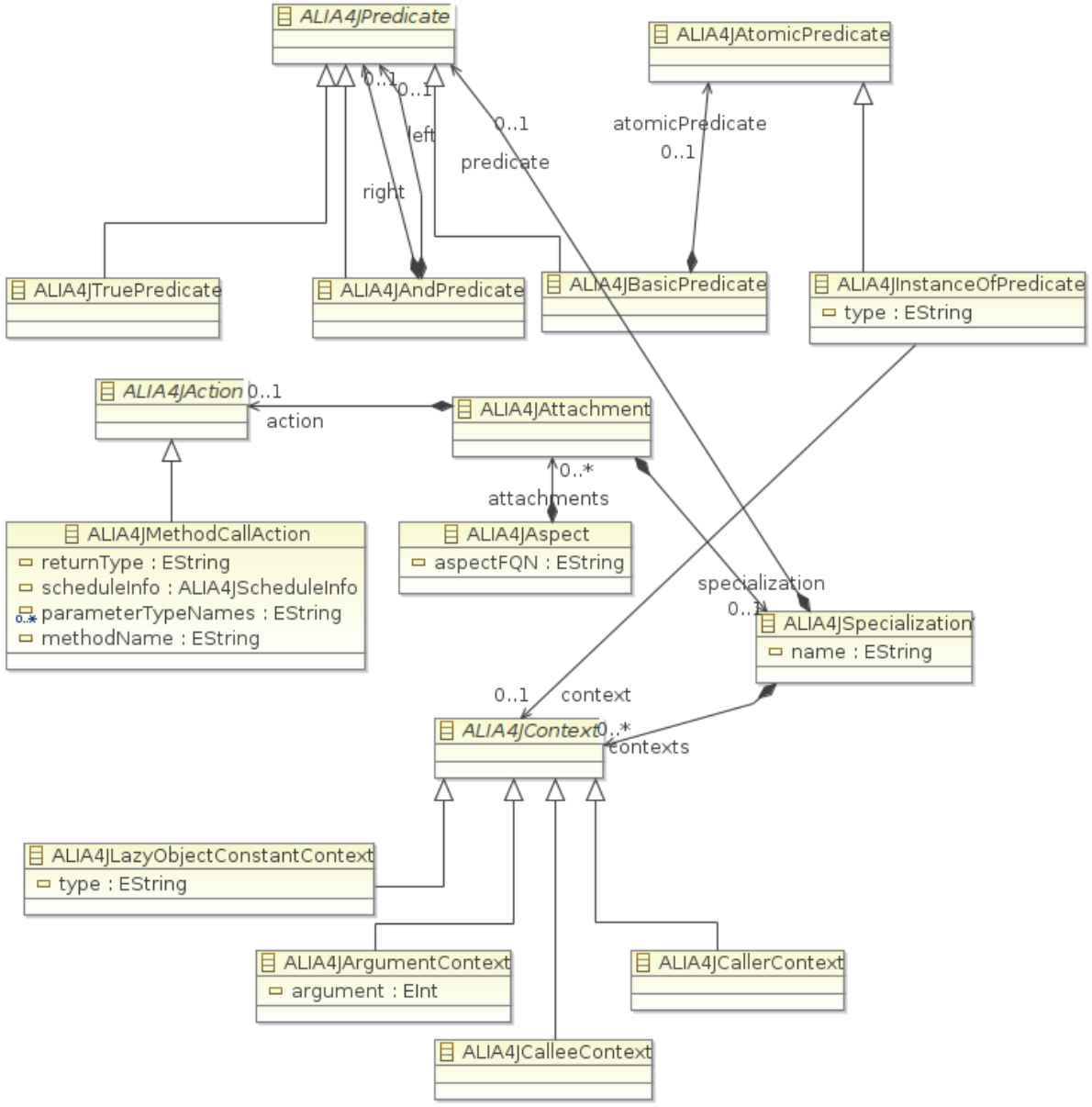

Figure 4.2: The ALIA4J meta-model with the Attachment, Actions, Contexts and Predicates.

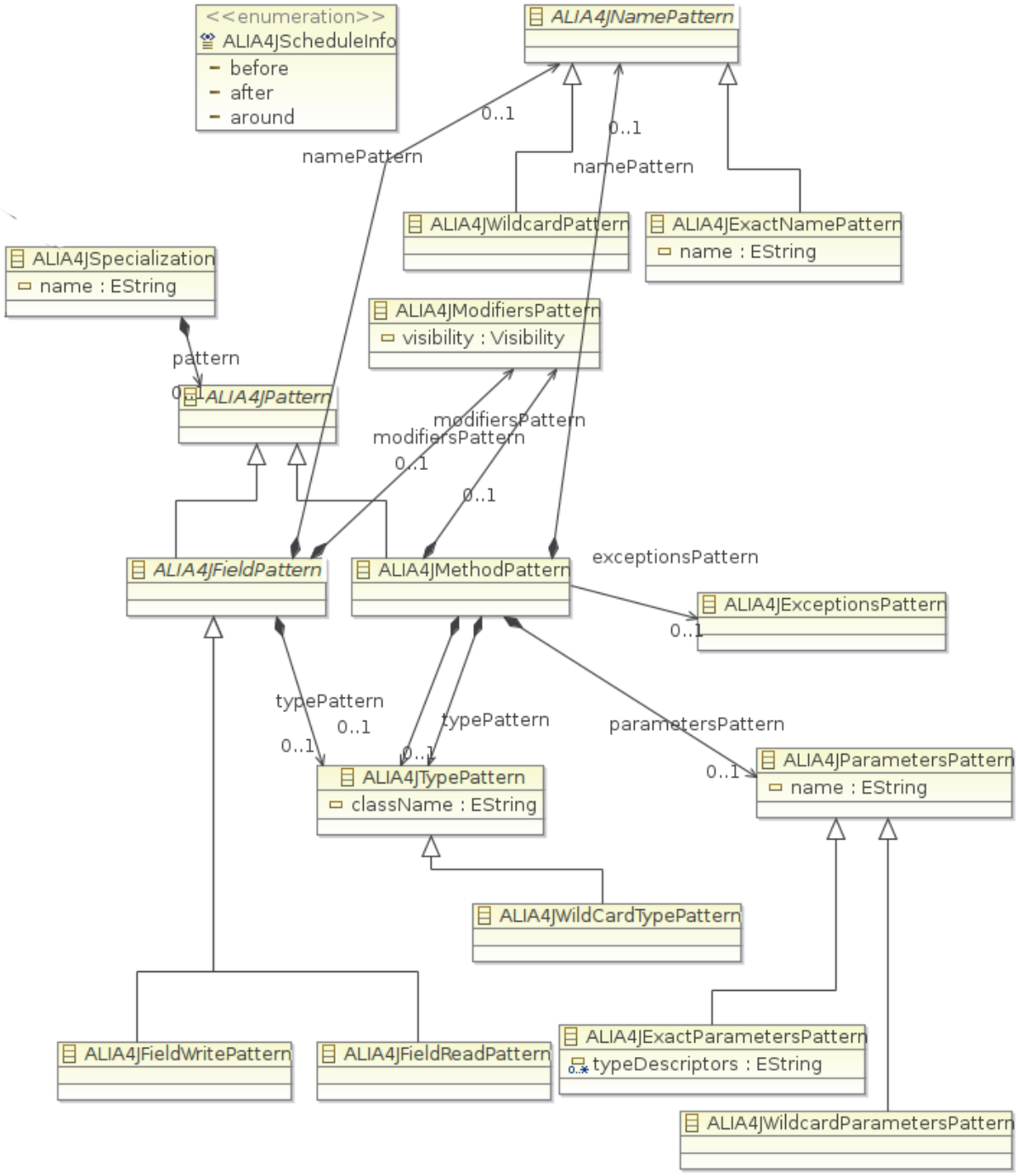

Figure 4.3: The ALIA4J meta-model containing the Patterns.

section. Two `ALIA4JSpecialization` objects with the same name and used in the same `ALIA4JAspect` are considered to be equal and will cause code for only one Specialization to be generated.

An alternative to this approach is to create and share one `ALIA4JSpecialization` between `ALIA4JAttachment` objects. Because a containment reference is used to reference the `ALIA4JSpecialization`, it is not allowed for a `ALIA4JSpecialization` to be contained by two different `ALIA4JAttachments`. To solve this problem, the reference between an `ALIA4JAttachment` and a `ALIA4JSpecialization` is changed to a *non-containment* reference. However this causes another problem because the `ALIA4JSpecialization` is now not contained by any entity. This requires another entity such as the `ALIA4JAspect` to contain the `ALIA4JSpecialization`. However this approach does not ideally resemble LIAM where a Specialization is only referenced by an Attachment.

Other meta-model classes are explained in more detail in section 2.3.1.
From the Ecore version of this meta-model, a *Genmodel* needs to be created using the eclipse wizard. This is a model that in addition to meta-model classes also contains information for generating the meta-model classes in Java. An example of this information is the path where the generated classes will be output. Figure 4.4 shows the generated ALIA4J Genmodel. The *Model Directory* property is set to `emf-gen` folder which means that the classes will be generated to this folder.

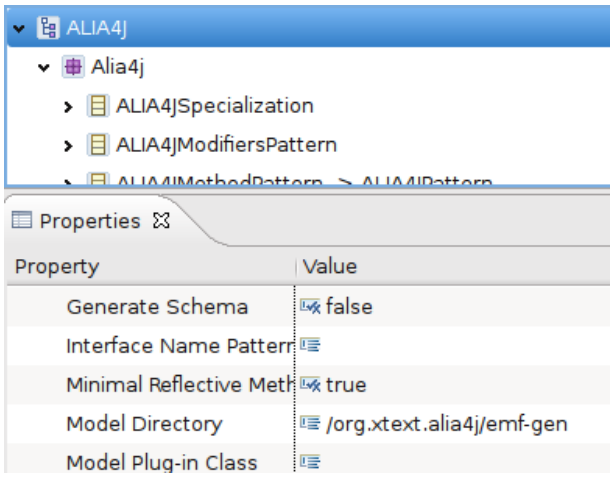

Figure 4.4: The generated ALIA4J Genmodel.

The Java classes that are generated from the meta-model using the Genmodel are organized in the following Java packages:

- `org.alia4j` - Interfaces and the Factory to create the Java classes.
- `org.alia4j.impl` - Concrete implementation of interfaces defined in `org.alia4j`.
- `org.alia4j.util` - The utility classes used by the EMF infrastructure.

Model Inference. Xtext would generate a skeleton class `ALIA4JJvmModelInferrer` if a language called *ALIA4J* is implemented. This skeleton class is therefore implemented and used to infer a Jvm model from the ALIA4J model that is later used for code generation. A class diagram is shown in figure 4.5. The `ALIA4JJvmModelInferrer` extends an abstract model inferrer that is provided by Xtext. A snippet of the `ALIA4JJvmModelInferrer` can be seen in Listing 4.1.

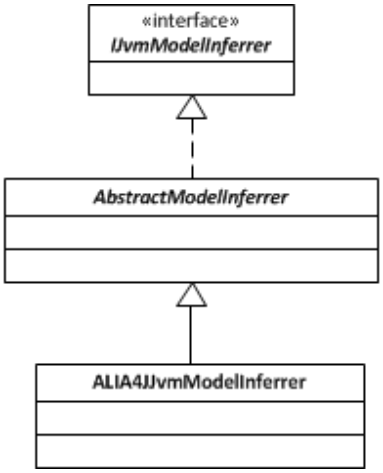

Figure 4.5: Class diagram of the `ALIA4JModelInferrer`

An `ALIA4JAspect` is transformed to a Java class that is used to setup the ALIA4J Attachments that describe the pointcut and advice from the aspect.
To transform a meta-model entity to a Java class in Xtext, infer methods are used where the first parameter represents the meta-model entity. The second parameter is of the type `IAcceptor`. This `IAcceptor` class is provided by Xtext. The `infer` method uses this class as a container to put all the resulting Java classes represented by `JvmDeclaredType` objects. The caller of the `infer` method uses this `IAcceptor` to access those objects. The `IAcceptor` is parametrized with `JvmDeclaredType` to specify that only `JvmDeclaredType` objects are allowed to be put into the `IAcceptor`. A `JvmDeclaredType` represents a Java class object in the Jvm model.

For illustration consider Listing 4.1 Line 1 where a Java class is inferred from an `ALIA4JAspect` by defining an `infer` method that takes an `ALIA4JAspect` as the first argu-

ment. Then in line 25, the `ALIA4JAspect` is transformed to a `JvmDeclaredType` by executing the `toClass` method. This is an extension method defined in the `JvmTypesBuilder` class. Extension methods are explained in section 3.3. The `toClass` method returns an initialized `JvmDeclaredType` object that represents a Java class for the aspect. The method is called on the `ALIA4JAspect` to specify that the `JvmDeclaredType` is the result of the `ALIA4JAspect`. The standard behaviour of Xtext is to create an association between the source model and the Jvm model. The source model is the ALIA4J model. Xtext uses the association internally to navigate between the two models. The `toClass` method takes the name of the class as a parameter. The name of this class is derived from the fully qualified name of the aspect concatenated with the text *Aspect*. The last argument of the `toClass` method is a closure. This argument is appended to the method call. A closure defines an anonymous function. For a detailed discussion on closures see section 3.3. The members that will be contained by the resulting Java class are added in this closure using the `members` property of the `JvmDeclaredType`. This property refers to a collection of all the Java methods and fields added to the class. These class members are discussed in the following paragraphs. The resulting `JvmDeclaredType` is then passed to the `accept` method of the `IAcceptor` class.

One of the class members is a method used to setup the ALIA4J Attachments. In line 28 this method is created with the name *setupAT*. One obvious approach that we did not take to organize the contents of this method is to include all Java code for the ALIA4J Attachments and Specializations in this method. This however will not allow us to deploy Attachments separately at different times when running the program. Although this is not a target in this thesis, it can be useful in future work. An aspect-oriented language may have a construct to arbitrarily enable or disable an advice based on some runtime condition.

The alternative approach that we take is to configure every Attachment in a separate method. The `setupAT` method then delegates to each of these methods. The inferrer loops through all the Attachments of the `ALIA4JAspect` in line 3. Then in line 13-21 a method is created for each attachment. These Java method objects are temporarily stored in a Java collection and are later, after looping through all the Attachments, added to the aspect class in line 27. The collection is per aspect and is accessed through the method `getMembersCollection`.

An ALIA4J Attachment contains a Specialization that corresponds to the pointcut to which an advice reacts. To allow an ALIA4J Specialization to be shared between Attachments in Java, it is not possible to generate Java code for the Specialization in the method of the Attachment. This prevents the same Specialization to be accessed by

multiple Attachments since every Attachment is configured in its own method. This is solved by creating a Java class field to store the Specialization. This field can then be accessed from multiple methods containing the Attachments. The Specialization is then configured in its own method.

As an illustration, lines 8-11 create a Java method for an `ALIA4JSpecialization`. This method object is temporarily stored in a Java collection and added to the aspect Java class in line 27. The name of this method is the name of the `ALIA4JSpecialization` concatenated with the text `_config`. The paragraph *Language Models* explained that two `ALIA4JSpecialization` objects with the same name will cause code to be created for only one `ALIA4JSpecialization`. This means that only one Java method for the Specialization will be added to the Java class instead of two Java methods for the two Specializations. This is accomplished by passing the name of the Java method for the Specialization to the collection when adding the Java method. If the collection already contains a Java method with that name, it will be replaced.
The Java class field that is used for the Specialization is created in line 7. The name of this field is determined by the name property of the `ALIA4JSpecialization`. This Java field is also added to the collection in the same way as the Java method. The name of the field is passed to the collection and if the collection already contains a Java field with that name, it will be replaced.

In this simplified inferrer, only MethodCallActions are supported. This is however not an inherent limitation of the inferrer and can be extended to support another Action. To support this, the meta-model needs to be changed to add an Action refinement such as the FieldWriteAction. This will be a subtype of `ALIA4JAction`. Then the `for` loop in line 4 needs to be adapted to check the type of the Action. The name of the method containing the attachment needs to be derived and the code for creating the Action and assigning it to the Attachment needs to be added to the body of the method.

Listing 4.1: Snippet from the ALIA4JJvmModelInferrer translating ALIA4J to Jvm

```
1  def dispatch void infer(ALIA4JAspect aspect, IAcceptor<JvmDeclaredType>
       acceptor, boolean prelinkingPhase) {
2
3    for(attachment : aspect.attachments) {
4          val action = attachment.action as ALIA4JMethodCallAction
5          val adviceMethodName = action.methodName
6          //Creating a field to hold the ALIA4J Specialization
7          getMembersCollection().put(attachment.specialization.name,
     attachment.specialization.toField(attachment.specialization.name,...
8          //Creating a Java method to configure the ALIA4J Specialization
9          val specMethodName = attachment.specialization.name + "_config"
```

```
10            getMembersCollection().put(specMethodName,
11              aspect.toMethod(specMethodName, ...
12            //Creating a Java method to configure the ALIA4J Attachment
13            val attachmentMethodName = adviceMethodName + "_attachment"
14            getMembersCollection().put(attachmentMethodName,
15            aspect.toMethod(attachmentMethodName, voidReturnType)[
16                  ...
17                  body = ['''
18                    Attachment attachment = new Attachment(...);
19                    org.alia4j.fial.System.deploy(attachment);
20                  ''']
21                ])
22          }
23      /* Creating a Java class containing ALIA4J Attachments and
            Specializations.
24       */
25      acceptor.accept(aspect.toClass(aspectFQN.toString + "Aspect") [
26        ...
27        members += getMembersCollection.values()
28        members += aspect.toMethod("setupAT", voidReturnType)[ ... ]
29      ])
30    }
```

Code generation. As discussed in the previous paragraph, a Jvm model is inferred from the ALIA4J model. From this Jvm model, Java code will be generated by the `JvmModelGenerator` that comes with Xtext. The `JvmDeclaredType` from the Jvm model results in a file containing a Java class definition. Xtext uses the folder `src-gen` as a default to contain the generated files. The `JvmDeclaredType` has a `name` property that contains the fully qualified name of the class. Both the name and the Java package of the generated class is derived from the value of the `name` property. The `JvmModelGenerator` generates the Java file in a location that reflects the Java package of the class. Consider as an example a fully qualified name of `x.y.Foo`. The `JvmModelGenerator` will generate a Java file called `Foo` in a folder `y` that is again contained in a folder `x`. This folder `x` is then contained in the default folder `src-gen`.

The Java class will contain Java methods and fields defined by the `members` property of the `JvmDeclaredType`. A `JvmOperation` object will result in a Java method and a `JvmField` results in a Java field. The name, type and visibility of this Java field are determined by properties of the `JvmField`. The name, visibility, parameters, return type and possible exceptions that can be thrown by the Java method are also defined by properties of the `JvmOperation`. The contents of the Java method is determined by a property called `body` of the `String` type. The `JvmModelGenerator` will copy the value of this property to the body of the Java method.

A security example. Consider as an example an ALIA4J model of a Security aspect. From this model a Jvm model will be inferred as discussed in the previous paragraphs. Assume that this Jvm model contains a `JvmDeclaredType` with a fully qualified name of *model.Security*. It contains a method for an Attachment and a method for a Specialization with the name *secureAccess*. A skeleton of the Java code that the code generator produces for this model is shown Listing 4.2. As described in the previous paragraphs, the `JvmDeclaredType` results in a `SecurityAspect` class definition that contains a method `setupAT` in line 19 that delegates to two other methods.
One is the method `secureAccess_specialization_config()` in line 5 to setup the Specialization. This Specialization is saved to the `secureAccess_specialization` Java field declared at line 3.
The other method `before_advice_method_secureAccess_attachment()` is declared in line 13 to setup and deploy the Attachment.

Listing 4.2: Java code skeleton resulting from an ALIA4J Jvm model

```
1  package model;
2  public class SecurityAspect {
3    private static Object secureAccess_specialization;
4
5    public static void secureAccess_specialization_config() throws
       ClassNotFoundException {
6
7      MethodPattern pattern_secureAccess_specialization = new MethodPattern
           (..); ...
8      secureAccess_specialization = new Specialization(
           pattern_secureAccess_specialization,
           predicate_secureAccess_specialization,
9      Arrays.asList(
10     ContextFactory.findOrCreateLazyObjectConstantContext(
           TypeHierarchyProvider.findOrCreateFromNormalTypeName("model.
           Security"))
11     ));
12   }
13   public static void before_advice_method_secureAccess_attachment() {
14     Attachment attachment = new Attachment(
15       Collections.singleton((Specialization)secureAccess_specialization),
16       ActionFactory.findOrCreateMethodCallAction(...), ...);
17     org.alia4j.fial.System.deploy(attachment);
18   }
19   public static void setupAT() throws ClassNotFoundException {
20   model.SecurityAspect.secureAccess_specialization_config();
21     model.SecurityAspect.before_advice_method_secureAccess_attachment();
22   }
23 }
```

4.2 The complete framework

This section covers the complete framework considering the development of an aspect-oriented language named *Alpha*. Xtext derives names of artefacts based on the name of the language that is developed.

Language models. The *Alpha meta-model* is generated by Xtext for the language *Alpha* as already explained in detail in section 3.1.

Model Inference. Section 3.1 described the use of the `AlphaJvmModelInferrer` to infer a Jvm model from the Alpha model. For the generic part of the framework, we require an additional ALIA4J model to be inferred besides a Jvm model. The ALIA4J model will be used by the language developer to specify ALIA4J concepts. The language developer does not have to consider ALIA4J API calls in Java. The `ALIA4JJvmModelInferrer` already developed and described in section 4.1, will infer a Jvm model from this ALIA4J model. Figure 4.7 shows the complete picture of model inference within the framework. The generic part of the framework has the responsibility of transforming the ALIA4J model into ALIA4J specific Java code. This part can be reused in any aspect-oriented language development project.

To enable the inferring of both the ALIA4J and Jvm models, an `AbstractALIA4JEnabledModelInferrer` has been developed that needs to be extended by the `AlphaJvmModelInferrer`. Ideally the `AlphaJvmModelInferrer` should be called `AlphaJvmAndALIA4JModelInferrer` however the skeleton class is generated by Xtext which is not ALIA4J-aware. The skeleton also does not extend the `AbstractALIA4JEnabledModelInferrer`.

Additionally, a helper class `ALIA4JTypesBuilder` has been developed, that provides extension methods used to generate instances of the ALIA4J meta-model. This is analogous to the `JvmTypesBuilder` provided by Xtext. The relationship between the classes is depicted in figure 4.6.

A skeleton of the Alpha-specific model inferrer is shown in Listing 4.3. Line 3 and 4 show how both the `JvmTypesBuilder` and `ALIA4JTypesBuilder` are injected. The `ALIA4JTypesBuilder` is added to the skeleton by the language developer.

Listing 4.3: Xtend code skeleton of the Alpha-specific model inferrer

```
1 class AlphaJvmModelInferrer extends AbstractALIA4JEnabledModelInferrer {
2
3   @Inject extension JvmTypesBuilder
4   @Inject extension ALIA4JTypesBuilder
5
6   def dispatch void infer(AlphaObject element, IAcceptor<ALIA4JAspect>
        acceptor) {
7     ...
8   }
9   def dispatch void infer(AlphaObject element, IAcceptor<JvmDeclaredType>
        acceptor,boolean isPrelinkingPhase) {
10    ...
11  }
12
13 }
```

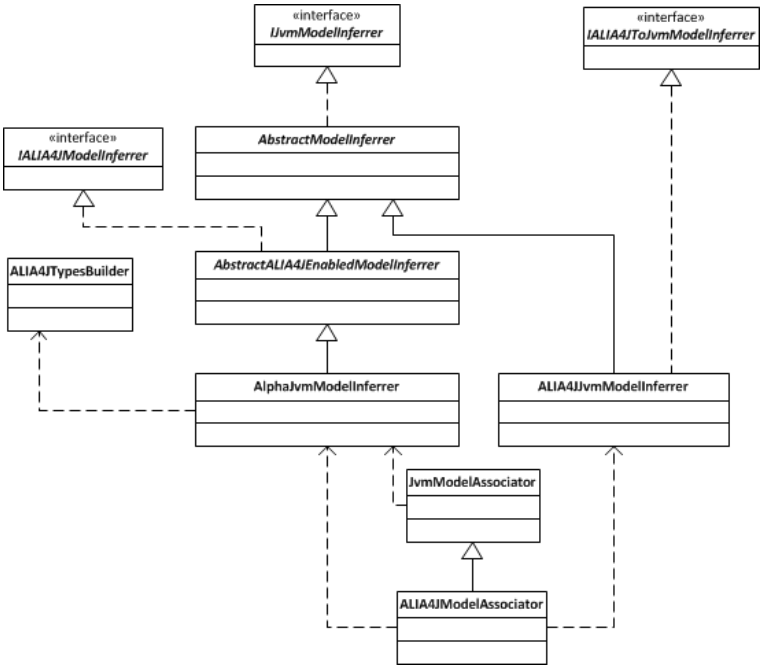

Figure 4.6: Class diagram of components in the Generic Framework

The skeleton is changed by the language developer to extend the class `AbstractALIA4JEnabledModelInferrer`. The language developer also needs to implement this skeleton to enable the `AlphaJvmModelInferrer` to infer both a Jvm and an ALIA4J model from a language model. The `AlphaJvmModelInferrer` implements two `infer` methods inherited from the `AbstractALIA4JEnabledModelInferrer`.

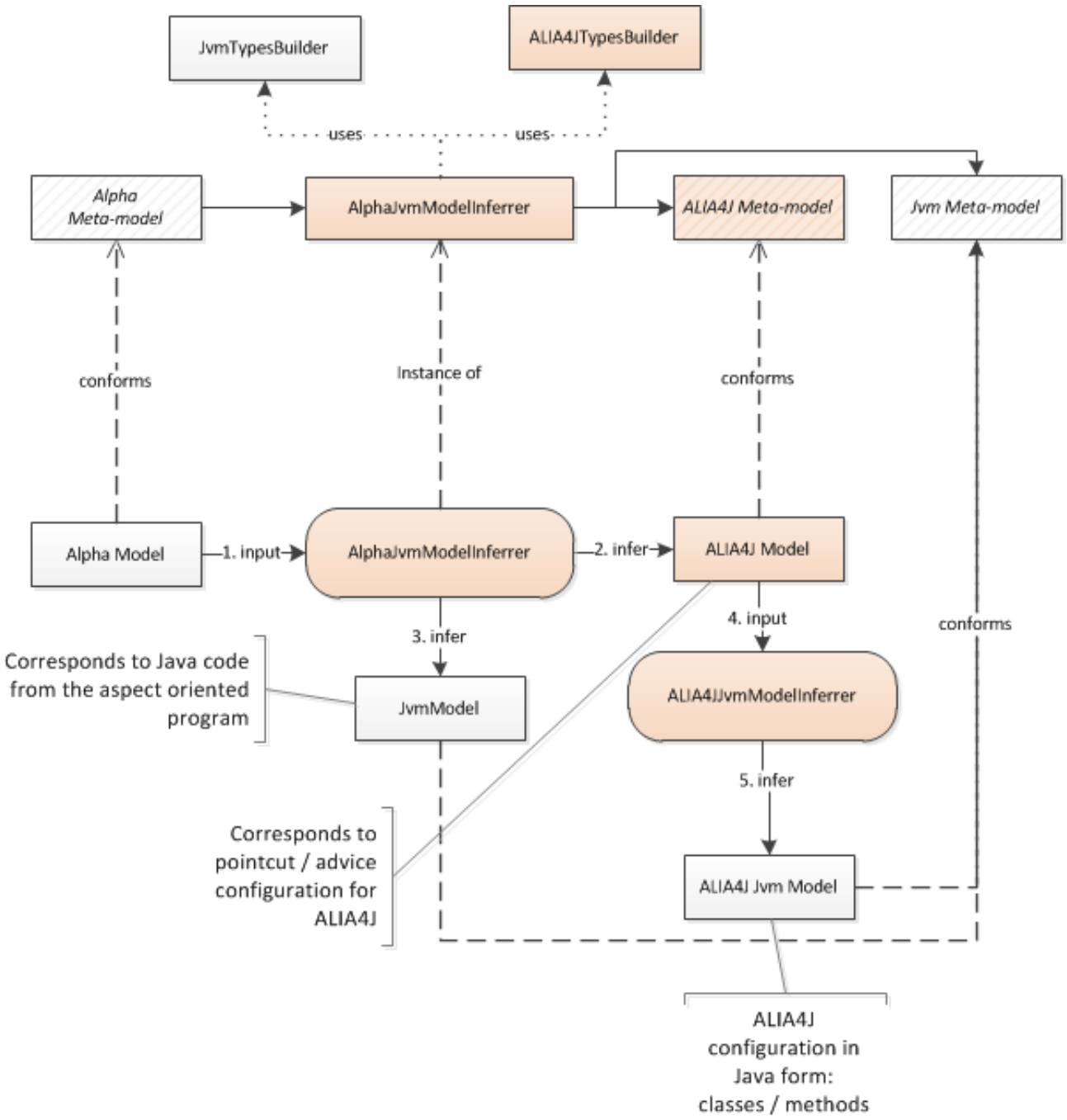

Figure 4.7: Inferring models in the Generic Framework

The first `infer` method in Line 6-8 is used to infer the ALIA4J model. The second parameter of type `IAcceptor` is used as a container to put all the resulting `ALIA4JAspect` objects. The caller of the `infer` method uses this `IAcceptor` to access the `ALIA4JAspect` objects. Type parameters are used to specify what type of objects are passed to the `IAcceptor`. In this case the `IAcceptor` is parametrized with `ALIA4JAspect`.
The second `infer` method in Line 9-11 is used to infer the Jvm model. In this case the `IAcceptor` is used to collect the `JvmDeclaredType` objects representing Java classes.

The `ALIA4JAspect` that is inferred using the first `infer` method needs to be input to the `ALIA4JJvmModelInferrer` as shown by the arrow number 4 in Figure 4.7. The mechanism by which this is accomplished is not shown in this Figure. However Figure 4.5 shows that the `AlphaJvmModelInferrer` and the `ALIA4JJvmModelInferrer` are both used by a class called `ALIA4JModelAssociator`. This class has been developed to execute both model inferrers by taking the `ALIA4JAspect` that is inferred by the `AlphaJvmModelInferrer` and executing the `ALIA4JJvmModelInferrer` to infer a `JvmDeclaredType` from the `ALIA4JAspect`. The `ALIA4JModelAssociator` does this by extending and overriding the `JvmModelAssociator`. The default behaviour of the `JvmModelAssociator` is to execute the `AlphaJvmModelInferrer` to infer a Jvm model from the Alpha model by calling the `infer` method on the model inferrer. The `ALIA4JModelAssociator` overrides this behaviour by calling the two `infer` methods on the `AlphaJvmModelInferrer` explained earlier. It then calls the `infer` method on the `ALIA4JJvmModelInferrer` using the resulting `ALIA4JAspect` as an argument to the method.

Code generation. Xtext only allows one implementation of the `IGenerator` interface to generate code. This is per default the `JvmModelGenerator` to generate Java code based on a Jvm model. Since we also want to be able to generate other types of code, a `CodeGenerator` class has been developed that will delegate to other implementations besides the `JvmModelGenerator`. This is shown in Figure 4.8. The `CodeGenerator` receives a resource containing all the generated models. This is shown by the top box in the figure. The `CodeGenerator` makes this resource available to the implementation it delegates to. It is up to this implementation to decide which code to generate for which model type.

The `CodeGenerator` delegates to the default `JvmModelGenerator` that will generate Java code for Jvm models. This is shown on the right side of the figure. The left side of the figure shows that the `CodeGenerator` also delegates to an `ALIA4JToXMIGenerator`. This is an implementation of the `CustomGenerator` interface that has been developed in this project. An implementation of this interface is injected through dependency injection. The `CodeGenerator` declares a field for the custom generator and the runtime

model configuration specifies which implementation should be injected as described in section 3.2.

The `ALIA4JToXMIGenerator` implementation is developed to generate an XMI (XML-based model interchange format) file for ALIA4J models. This XMI file is helpful by enabling to inspect the ALIA4J model visually in eclipse and to see whether it is valid. Listing 4.4 shows a snippet of the runtime model to specify that a `ALIA4JToXMIGenerator` implementation should be used. This allows the `CodeGenerator` and the `ALIA4JToXMIGenerator` to be completely decoupled from each other. This also creates the possibility to inject another class to generate other type of code.

Listing 4.4: Specifying a CustomGenerator implementation in the AlphaRuntimeModel

```
public Class<? extends org.xtext.alia4j.CustomGenerator>
    bindCustomGenerator() {
  return org.xtext.alia4j.ALIA4JToXMIGenerator.class;
}
```

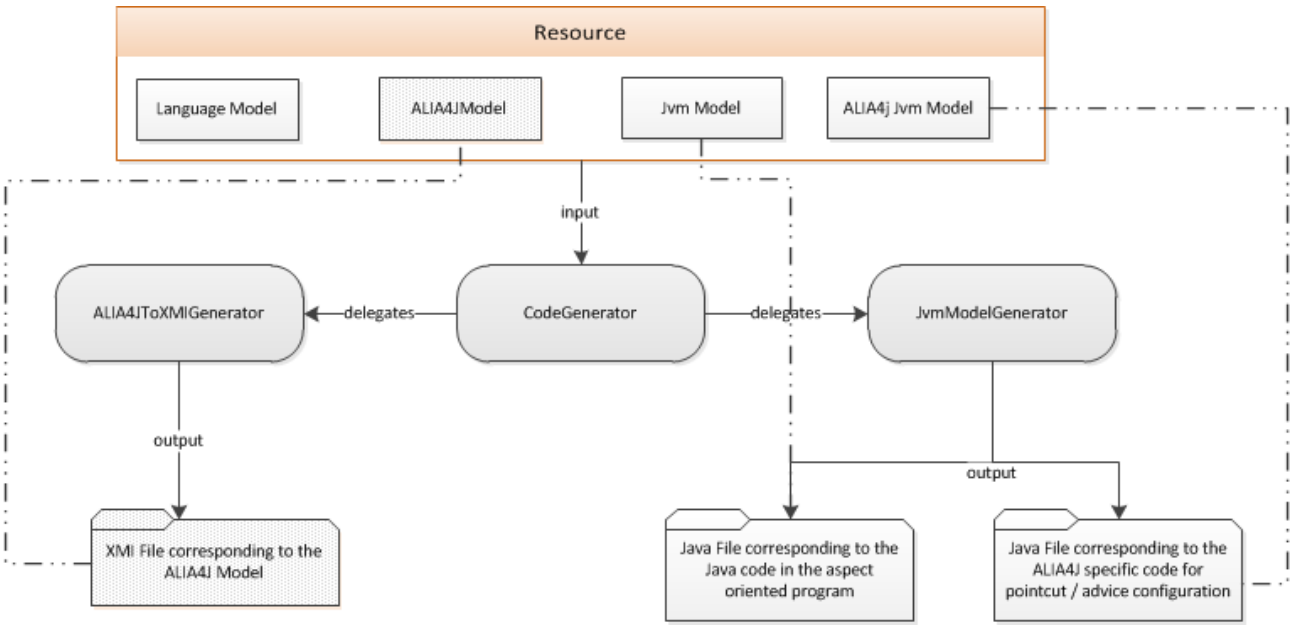

Figure 4.8: Generating code in the Generic Framework

4.3 Extensibility

When implementing a new language following the ALIA4J approach, the semantics of the language must be described by either re-using existing meta-model refinements, implementing new refinements, or a mixture of both. Suppose that we want to implement a new refinement of Predicate to express the control flow construct from AspectJ. We assume that ALIA4J already contains an implementation of the CFlowPredicate entity that does not yet exists in the Ecore meta-model. To add this refinement, there are several solutions. This section describes three solutions for adding LIAM refinements by either creating subclasses or changing the framework to make it more extensible.

Adding a meta-model class. In both the solutions described in section 4.3.1 and 4.3.2, a new LIAM entity is added by changing the Ecore meta-model. To support the control flow construct from AspectJ a new predicate `ALIA4JCFlowPredicate` is created as a subclass of the already provided `ALIA4JPredicate`. After this the meta-model classes need to be re-generated as described in 4.1.

Extending the `ALIA4JTypesBuilder`. The `ALIA4JTypesBuilder` that is used to instantiate and initialize ALIA4J types defined in the Ecore version of LIAM does not know about a new type. One can create a subclass of the `ALIA4JTypesBuilder` with an additional method to create and initialize the `ALIA4JCFlowPredicate`. The `ALIA4JTypesBuilder` in Line 4 Listing 4.3 will be replaced by this subclass so the language specific model inferrer can make use of the new extension method.
It is however not required to do this since the `ALIA4JTypesBuilder` is only used to make the code of the model inferrer more concise. An alternative is to initialize the ALIA4J type directly in the model inferrer.

4.3.1 Solution by extending the `ALIA4JJvmModelInferrer`

The current implementation of the `ALIA4JJvmModelInferrer` infers Java code specific to every LIAM entity implementation. Listing 4.5 shows that there exists a specific `generateALIA4JPattern` and `generateALIA4JPredicate` method for every specific ALIA4J Pattern and Predicate such as `ALIA4JInstanceOfPredicate` and `ALIA4JMethodPattern`. The appropriate method is executed for every specific ALIA4J Pattern or Predicate. These are passed as an argument to the method. The methods are used to generate code that will be put in the body of the Java method used for the ALIA4J Specialization. By adding a `generateALIA4JPredicate` method specific for the

ALIA4JCFlowPredicate, this method will automatically be used to infer Java code when a ALIA4JCFlowPredicate is encountered.

Listing 4.5: ALIA4JJvmModelInferrer with methods for Predicate implementations

```
1  def dispatch generateALIA4JPredicate(ALIA4JTruePredicate p, String
       varName)
2    ...
3  def dispatch generateALIA4JPredicate(ALIA4JInstanceOfPredicate p, String
        varName)
4    ...
5  def dispatch generateALIA4JPattern(ALIA4JMethodPattern pattern, String
       varName)
6    ...
```

Since the ALIA4JJvmModelInferrer does not know what to infer from a new ALIA4JCFlowPredicate, this means that the ALIA4JJvmModelInferrer needs to be changed to infer Java code from the new LIAM entity. It is obvious that this is not the best approach since it requires to change and recompile the model inferrer provided by the framework.

An alternative is to extend the ALIA4JJvmModelInferrer by creating a subclass CustomALIA4JJvmModelInferrer as can be seen in Line 1 of Listing 4.6. Methods inherited from the ALIA4JJvmModelInferrer can be used to generate other code. An example is the generateALIA4JSpecializationCode method that is used in Line 9 to generate code for every ALIA4J Specialization contained in the ALIA4JCFlowPredicate.

Listing 4.6: The CustomALIA4JJvmModelInferrer subclass

```
1  class CustomALIA4JJvmModelInferrer extends ALIA4JJvmModelInferrer {
2    def dispatch generateALIA4JPredicate(ALIA4JCFlowPredicate p, String
         varName) {
3    '''
4      Set<Specialization> col = new HashSet<Specialization>();
5      «var i = -1»
6      «FOR spec: p.specializations»
7        «spec.name = "spec"+(i = i + 1)»
8        Specialization «spec.name»;
9        «spec.generateALIA4JSpecializationCode()»
10       col.add(«spec.name»);
11     «ENDFOR»
12       Predicate<AtomicPredicate> «varName» = new BasicPredicate<
             AtomicPredicate>(
13         AtomicPredicateFactory.findOrCreateCFlowPredicate(col),true
14       );
15   '''
```

```
16    }
17 }
```

For this custom class to be used instead of the default `ALIA4JJvmModelInferrer`, it has to be specified in the `AlphaRuntimeModule` described in section 3.2. Listing 4.7 depicts this by returning the `CustomALIA4JJvmModelInferrer` class.

Listing 4.7: Binding the `CustomALIA4JJvmModelInferrer` in the `AlphaRuntimeModule`

```
public Class<? extends IALIA4JToJvmModelInferrer>
    bindIALIA4JToJvmModelInferrer() {
     return CustomALIA4JJvmModelInferrer.class;
   }
```

4.3.2 Solution by delegating Java code inference to ALIA4J meta-models

Another solution is to change the framework in such a way to incorporate the inference of Java code in the ALIA4J meta-model entity itself, instead of in the model inferrer. The `ALIA4JJvmModelInferrer` will delegate the inference by calling a special infer method on the meta-model entity. This approach allows the `ALIA4JJvmModelInferrer` to only be aware of abstract ALIA4J entities and eliminates the need for changing or extending the `ALIA4JJvmModelInferrer`. This will increase the extensibility of the generic framework. Adding a custom implementation of a ALIA4J meta-model entity will be enough.

This is demonstrated by adding a method `String inferJava(String varName)` to the abstract `ALIA4JPredicate` and `ALIA4JPattern`. Every specific entity will provide an implementation of this method. The implementation for the `ALIA4JCFlowPredicate` is shown in Listing 4.8 which is analogous to the `generateALIA4JPredicate` method from Listing 4.6. However the inference of Java code for the specialization is delegated to the `ALIA4JSpecialization` by calling the `inferJava` method in Line 9.

Listing 4.8: The inferJava method implementation for the ALIA4JCFlowPredicate

```
1  public String inferJava(String varName) {
2    StringConcatenation _builder = new StringConcatenation();
3    _builder.append("Set<Specialization> col = new HashSet<Specialization
         >();");
4    _builder.newLine();
5    int i = 0;
6    for (ALIA4JSpecialization spec: getSpecializations()) {
7      spec.setName("spec"+i++);
```

```
8         _builder.append("Specialization " + spec.getName());
9         _builder.append(spec.inferJava());
10        _builder.append("col.add(" + spec.getName() + ")");
11      }
12      _builder.append(String.format("Predicate<AtomicPredicate> %s = new
            BasicPredicate<AtomicPredicate>(",varName));
13      _builder.append(" AtomicPredicateFactory.findOrCreateCFlowPredicate(
            col),true");
14      _builder.append(");");
15
16      return _builder.toString();
17    }
```

The meta-model Java classes are generated from the Ecore model. It is possible to change the generated code but this code will get lost when the classes are re-generated. To prevent this, the`@Generated` annotation of the class or method that contains the changes needs to be removed. It is however easy to forget to remove this. It is obvious that this is not the best way to implement changes.
A better idea is to implement the `inferJava` method separately from the generated class. This is possible by extending the generated class. However this is still a problem because the generated code does not know about the subclass. The code generated by EMF contains a factory class that instantiates the individual classes. The factory class generated for the ALIA4J meta-model is shown in Listing 4.9. The factory refers to other generated implementations such as the `ALIA4JInstanceOfPredicateImpl` in Line 16 and not to a possible subclass. This still forces us to change the generated code.

Listing 4.9: The generated Alia4jFactoryImpl implementation

```
1  /**
2   * <!-- begin-user-doc -->
3   * An implementation of the model <b>Factory</b>.
4   * <!-- end-user-doc -->
5   * @generated
6   */
7  public class Alia4jFactoryImpl extends EFactoryImpl implements
       Alia4jFactory
8  {
9    /**
10    * <!-- begin-user-doc -->
11    * <!-- end-user-doc -->
12    * @generated
13    */
14   public ALIA4JInstanceOfPredicate createALIA4JInstanceOfPredicate()
15   {
16     ALIA4JInstanceOfPredicateImpl alia4JInstanceOfPredicate = new
           ALIA4JInstanceOfPredicateImpl();
```

```
17      return alia4JInstanceOfPredicate;
18    }
19 ...
20 }
```

A solution is to allow the class generator to know about subclasses and use them automatically in the generated code. Xtext provides an `EcoreGenerator` that supports this. The source path property of this generator can be set to the folder that will be scanned to find subclasses. This `EcoreGenerator` is used in the workflow configuration in Listing 4.10. This configuration allows to declare object instances, attribute values and references. Line 13 declares an instance of a component that cleans the `emf-gen` directory where the generated classes will be output. Line 17 declares an `EcoreGenerator` instance that will generate the meta-model classes. The `srcPath` property is set to where the custom classes can be found and the `genModel` property to the generator model used to generate classes. When running this Workflow, the `EcoreGenerator` instance will search for classes that extend one of the meta-model classes. The `EcoreGenerator` requires those classes to have the name of the meta-model class postfixed with the text *Custom*. This cannot be configured. A solution for allowing a different name is to subclass the `EcoreGenerator`. An example of a custom class is the `ALIA4JCFlowPredicateImplCustom` that extends `ALIA4JCFlowPredicateImpl`. If the meta-model classes are already generated when adding the subclass, the Workflow should be run again. This Workflow is defined in a file called `alia4j.mwe2` shown in Listing 4.10 which is part of our framework.

Listing 4.10: The alia4j Workflow definition file

```
1  module org.eclipse.xtext.alia4j
2
3  import org.eclipse.emf.mwe.utils.*
4
5  var projectName = "org.xtext.alia4j"
6  var runtimeProject = "${projectName}"
7
8  Workflow {
9    bean = StandaloneSetup {
10     platformUri = ".."
11   }
12
13   component = DirectoryCleaner {
14     directory = "${runtimeProject}/emf-gen"
15   }
16
17   component = org.eclipse.emf.mwe2.ecore.EcoreGenerator {
18     srcPath = "platform:/resource/${runtimeProject}/src"
```

```
19      genModel = "platform:/resource/${runtimeProject}/model/ALIA4J.genmodel
            "
20    }
21  }
```

4.3.3 Solution by using a different ALIA4J Ecore meta-model

Another solution is to make use of the naming conventions used in the ALIA4J API to create specific entities of ALIA4J models. The API contains factory methods to create entities. An example of a factory method to create a CFlowPredicate looks like this: `AtomicPredicate findOrCreateCFlowPredicate(final Set<Specialization> specializations)`. The name of the method is always the text *findOrCreate* followed by the name of the predicate concatenated with the text *Predicate*. Making use of this information, a different implementation can be used for the ALIA4J Ecore meta-model. Instead of having a specific meta-model class for each possible entity refinement, one class is used with the following properties:

- The factory name. This is in the example an `AtomicPredicateFactory`.
- The predicate name such as the `CFlow` or `Instanceof` predicate.
- A map to contain the name and value of each parameter that needs to be passed to the factory method. In the example of the `CFlow` predicate this is a collection of Specializations. For an `Instanceof` predicate this is a Context and TypeDescriptor.

The intended generic Ecore model is visualized in Figure 4.9. The `ALIA4JPattern`, `ALIA4JPredicate`, `ALIA4JContext` and `ALIA4JAction` entities have the properties defined in the previous list. One can see that the specializations of these entities have been removed if compared with the model described in section 4.1.

This meta-model makes it possible to implement a generic code generation mechanism that will generate code for specific LIAM entities. In addition to the properties in the previous list, the code generation depends on the type of the values in the map such as Context, TypeDescriptor or a collection of Specializations. This makes it more complicated because of the need for more generic support. A type-check needs to be done on those values to determine what kind of Java code will be generated.
What makes it even more complicated is that from the name of a property in the parameter map, the parameter position in the factory method has to be inferred. This is not

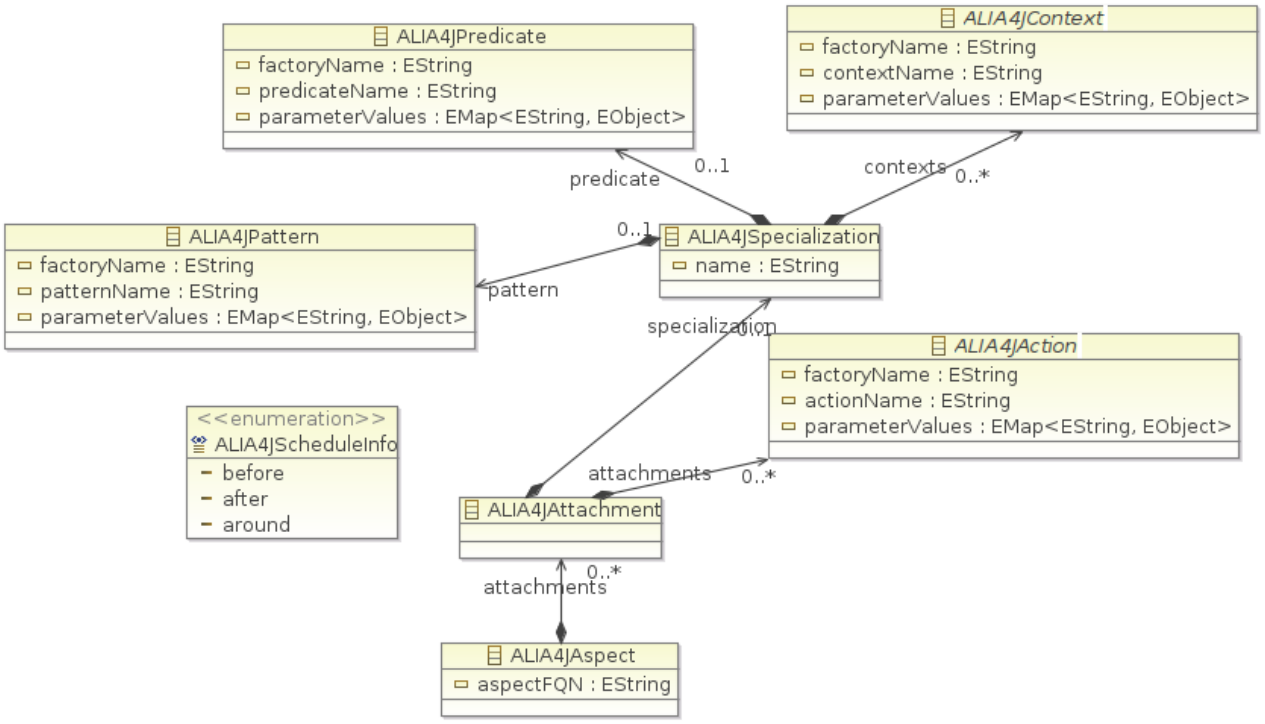

Figure 4.9: The intended ALIA4J meta-model when using the API naming conventions

possible using Java reflection. A solution to this is by explicitly annotating the factory method to specify the parameter index for the parameter name. Reflection will be used to retrieve the parameter position and the code generator can generate the right code for every parameter position of the factory method.

4.3.4 Comparison

In this section we compare the solutions presented in the previous sections by examining the required changes to the framework when adding new LIAM entities. This comparison is visualized in the table below.

In the first solution there is a need to change the ALIA4J Ecore meta-model for every new LIAM entity. Then the model inferrer provided by the framework needs to be extended for it to infer the Jvm model from the new entity. This does not scale very well because the framework needs to be recompiled every time a new LIAM entity is added.

The second solution offloads the code generation to the provider of the ALIA4J meta-model entity. However this approach requires a custom implementation for every a new LIAM entity. The meta-model Java classes need to be generated again for the framework to know about the custom implementation. This also does not scale very well.

The third solution requires no implementation for a new LIAM entity and classes provided by the framework do not have to be changed and recompiled. This makes the framework very scalable.

Table 4.1: Comparison table for the three presented solutions

Solution	Changes when adding new LIAM entity
By extending the `ALIA4JJvmModelInferrer`	add new Ecore entity to the framework + add subclassed ALIA4J model inferrer to language implementation
By delegating Java inference to meta-model entity	add new Ecore entity to the framework + add subclassed entity to language implementation
By using a different ALIA4J Ecore meta-model	-

5

Evaluation

5.1 AspectJ Compiler Implementation

The generic compiler framework is validated by developing an implementation for the aspect-oriented language AspectJ. To demonstrate a clear separation between this language specific implementation and the generic framework, these are implemented in different eclipse projects.

Language models. An Xtext grammar `AspectJ.xtext` is developed to define the language constructs. An AspectJ meta-model is derived from this Xtext grammar. Part of the meta-model is depicted in Figure 5.1. The `CompilationUnit` meta-model class contains type definitions represented by the `TypeDefinition` class. Those type definitions can be a `JavaClassDefinition`, `JavaInterfaceDefinition` or an `AspectDefinition`. A `TypeDefinition` specifies several properties like the name of the type and the visibility such as `public` or `private`. A type contains members such as methods and fields that are respectively modelled by the `JavaMethodDefinition` and `JavaFieldDeclaration` classes. These inherit from the `JavaMember` class. The `AspectMember` class models members from aspects such as advice and pointcut declarations. These are modelled using the `AspectAdviceDeclaration` and `AspectpointcutDeclaration` classes. A `JavaMember` can also be an `AspectMember` because aspects can contain Java methods and fields. `AspectMember` therefore inherits from `JavaMember`.

Model Inference. Xtext generated a `AspectJJvmModelInferrer` skeleton class. This class is changed to extend the `AbstractALIA4JEnabledModelInferrer` to be able to infer both Jvm and ALIA4J models.

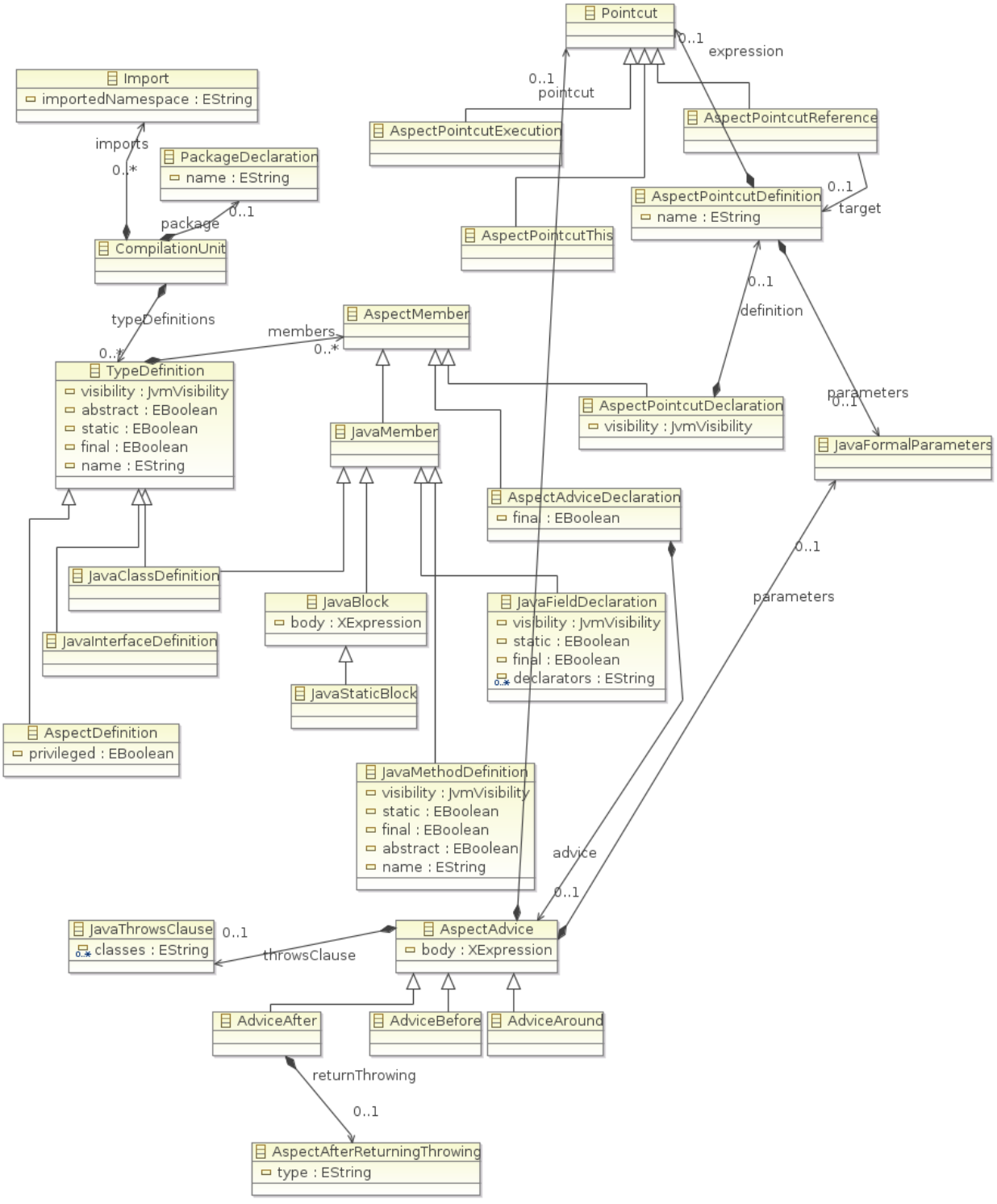

Figure 5.1: Part of the AspectJ meta-model

The inferring of models when compiling an AspectJ program is demonstrated in Figure 5.2. Part of the implemented `AspectJJvmModelInferrer` is shown in Listing 5.1. As already described in section 4.1 and shown in Listing 4.3, there are two `infer` methods. The first method from Listing 5.1 lines 10-49 shows the inferring of an `ALIA4JAspect` from an `AspectDefinition`. The second method from lines 50-98 show the inferring of a `JvmDeclaredType` from an `AspectDefinition`.

The first method loops through the members of an `AspectDefinition` such as an `AspectAdviceDeclaration` or an `AspectVariableDeclaration`. The type of member is determined by the Xtend `switch` statement. For every `AspectAdviceDeclaration` an `ALIA4JAttachment` is inferred in lines 20-42. This is done using the `toALIA4JAttachment` method from line 22. This method is an extension method provided by the `ALIA4JTypesBuilder`. The `ALIA4JSpecialization` is inferred in line 23 through delegation to the method `inferSpecialization`.
Lines 24-41 then infers an `ALIA4JMethodCallAction`. The parameter types of the method that is called by this action are set in Line 27. The return type of this method is set in line 32. The ScheduleInfo to specify when the action will call the method is set in lines 33-39. Both the specialization and action property of the Attachment are set in line 23 and 24.

The second method from lines 50-98 show the inferring of a `JvmDeclaredType` from an `AspectDefinition`. This method also loops through the members of an `AspectDefinition`. However since an aspect can contain normal Java fields and methods beside an advice, these are also taken into account when looping through the members. In lines 67-78 a `JvmOperation` and `JvmField` are inferred from the `JavaMethodDefinition` and `JavaFieldDeclaration` in the AspectJ model. In contrast to inferring an `ALIA4JAttachment` from an `AspectAdviceDeclaration` in line 22, a `JvmOperation` is inferred in line 86. This will contain the Java code from the advice. The name of this advice Java method must be consistent with the name used in the MethodCallAction of the Attachment. This cannot be ensured by the generic framework since the advice Java method is not created by the framework but by the language specific model inferrer. One solution that we did not take is for the language developer to create a method in the model inferrer to derive this name. This method will then be called from two different places namely when inferring the `ALIA4JMethodCallAction` and when inferring the advice Java method.

To demonstrate an alternative approach, Lines 21 and 85 show how this is ensured by allowing a custom meta-model class `AspectAdviceImplCustom` for the advice AST node to infer the method name for the advice. This is similar to the approach discussed in section 4.3.2 where a subclass of the meta-model entity is used to infer Java code. In

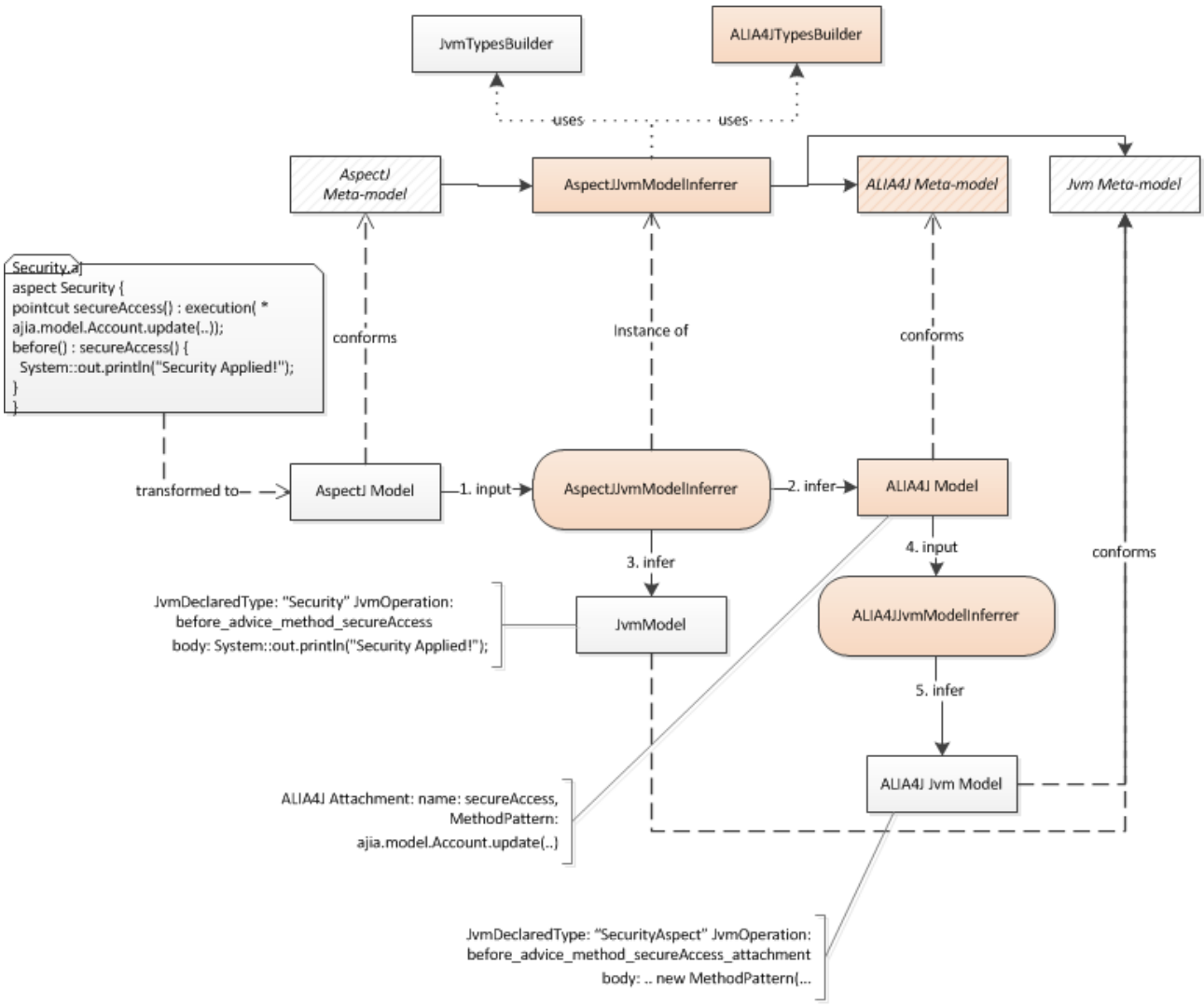

Figure 5.2: Inferring models in the AspectJ Compiler

this case the custom class contains a method `inferAdviceMethodName` to generate the name for the advice method. This always generates the same name and is called by the inferrer whenever this is required. The custom class is created by the language developer to extend the advice meta-model class and is shown in Listing 5.2. Custom meta-model classes can also be used to store information during the first step of inferring an ALIA4J model. This information can then be accessed in the next step where the Jvm model is inferred from the AspectJ model. By creating a class with the name of the meta-model class postfixed with the text *Custom*, Xtext will use the custom class as a substitute of the generated meta-model class.

Listing 5.1: The `AspectJJvmModelInferrer` inferring Jvm and ALIA4J models

```
1  class AspectJJvmModelInferrer extends AbstractALIA4JEnabledModelInferrer {
2
3
4    @Inject extension JvmTypesBuilder
5    @Inject extension TypeReferences typeReferences
6    @Inject extension IQualifiedNameProvider
7
8    @Inject extension CustomALIA4JTypesBuilder
9
10   /**
11    * Infer an ALIA4JAspect from an AspectDefinition
12    */
13   def dispatch void infer(AspectDefinition element, IAcceptor<ALIA4JAspect
       > acceptor) {
14     val aspectName = element.fullyQualifiedName.toString
15     acceptor.accept(element.toALIA4JAspect() [
16       aspectFQN = aspectName
17       for (member : element.members) {
18         switch member {
19           AspectAdviceDeclaration : {
20             val pct = member.advice.pointcut as AspectPointcutReference
21             val adviceMethodName = (member.advice as
     AspectAdviceImplCustom) .inferAdviceMethodName()
22             attachments += element.toALIA4JAttachment() [
23                 specialization = inferSpecialization( pct.target ,
     aspectName)
24               action = element.toALIA4JMethodCallAction() [
25                 methodName = adviceMethodName
26                 for(p : member.advice.parameters.parameters) {
27                   parameters += p.toALIA4JParameter(p.name, p.
     parameterType.simpleName)
28                 }
29                 val advice = member.advice
30                 switch advice {
31                   AdviceAround :{
32                     returnType = advice.returnType.toString
```

```
33                    scheduleInfo = ALIA4JScheduleInfo::^AROUND
34                  }
35                  AdviceBefore :{
36                    scheduleInfo = ALIA4JScheduleInfo::^BEFORE
37                  }
38                  AdviceAfter :{
39                    scheduleInfo = ALIA4JScheduleInfo::^AFTER
40                  }
41                }
42              ]
43
44          ]
45          }
46        }
47      }
48  ])
49  }
50  /**
51   * Infer Aspect
52   */
53  def dispatch void infer(AspectDefinition element, IAcceptor<
        JvmDeclaredType> acceptor,boolean isPrelinkingPhase) {
54    val JvmTypeReference voidReturnType = typeReferences.getTypeForName(
        typeof(void) as Class, element)
55    val aspectFQN = element.fullyQualifiedName
56    val aspectStaticMembers = new LinkedList<JvmMember>()
57
58    acceptor.accept(element.toClass(aspectFQN) [
59    documentation = element.documentation
60
61    var JvmTypeReference alia4jSpecializationType = element.newTypeRef(
        typeof(Object))
62    var List<String> methodcalls = new LinkedList();
63
64    for (member : element.members) {
65      switch member {
66        JavaMethodDefinition :{
67          members += member.toMethod(member.name, member.returnType)[
68            for(p : member.parameters) {
69              parameters += p.toParameter(p.name, p.parameterType)
70            }
71            ^static = member.^static
72            documentation = member.documentation
73            it.body = member.body
74          ]
75        }
76        JavaFieldDeclaration : {
77          for(id : member.declarators)
78            members += member.toField(id, member.type) [
79              ^static = member.^static
80              visibility=visibility
81            ]
82        }
```

```
83          AspectAdviceDeclaration : {
84            val pct = member.advice.pointcut as AspectPointcutReference
85            val _adviceMethodName = (member.advice as AspectAdviceImplCustom
        ).inferAdviceMethodName()
86            members.add(member.toMethod(_adviceMethodName,voidReturnType)[
87                  for(p : member.advice.parameters.parameters) {
88                    parameters += p.toParameter(p.name, p.parameterType)
89                  }
90                  documentation = member.documentation
91                  final = member.^final
92                  body = member.advice.body
93                ])
94          }
95        }
96      }
97      ])
98    }
99    def ALIA4JSpecialization inferSpecialization(Pointcut pointcut, String
          aspectFQN) {
100     val PredicatePatternAndContext result =
            inferPredicatePatternAndContext(pointcut,aspectFQN)
101     val List<ALIA4JContext> _context = new LinkedList()
102     //Use a LazyObjectConstantContext to instantiate the advice class
            containing the advice method
103     _context += pointcut.toALIA4JLazyObjectConstantContext(aspectFQN)
104     _context += result.contexts
105     return pointcut.toALIA4JSpecialization() [
106       pattern = result.pattern
107       predicate = result.predicate
108       contexts += _context
109     ]
110   }
111   def dispatch PredicatePatternAndContext inferPredicatePatternAndContext(
          AspectPointcutExecution pct, String aspectFQN ) {
112     var ALIA4JPattern pattern = inferALIA4JMethodPattern(pct)
113     return new PredicatePatternAndContext(pct.toALIA4JTruePredicate(),
            pattern, new LinkedList())
114   }
115   def dispatch PredicatePatternAndContext inferPredicatePatternAndContext(
          AspectPointcutThis pct, String aspectFQN ) {
116
117     val context = pct.toALIA4JCallerContext()
118     val target = pct.target as FullJvmFormalParameter
119     val _predicate = pct.toALIA4JInstanceOfPredicate(context, target.
            parameterType.qualifiedName)
120
121     return new PredicatePatternAndContext(_predicate, null, context)
122   }
123   ....
124 }
```

Listing 5.2: A custom AST node for the Aspect Advice

```
public class AspectAdviceImplCustom extends AspectAdviceImpl {
  public String inferAdviceMethodName() {
    AspectPointcutReference pct = (AspectPointcutReference) getPointcut();
    return "advice_method_" + pct.getTarget().getName();
  }
}
```

Supporting cflow. The ALIA4J Ecore meta-model did not have an `ALIA4JCFlowPredicate` to use for the `cflow` construct of AspectJ. To support this, the solution described in section 4.3.1 was used. A `CustomALIA4JTypesBuilder` has been implemented and a subclass of the `ALIA4JJvmModelInferrer` has been developed. The `CustomALIA4JTypesBuilder` is injected in line 8 of Listing 5.1.

Listing 5.1 line 100 shows that the `inferSpecialization` uses the method `inferPredicatePatternAndContext` to infer a predicate, pattern and context for every type of AspectJ pointcut expression construct such as `execution` and `this`. Which `inferPredicatePatternAndContext` method is executed depends on the expression type. By adding a new method for the `cflow` construct shown in Listing 5.3, one can infer the right ALIA4J entities needed. The `toALIA4JCFlowPredicate` extension method from `CustomALIA4JTypesBuilder` is used in line 2. Since the `ALIA4JCFlowPredicate` takes a set of Specializations as parameter, the `inferSpecialization` method is used to infer a Specialization from the expression within the `cflow` construct. This example however is simplified by allowing only one Specialization.

Listing 5.3: Inferring an added cflow predicate

```
1  def dispatch PredicatePatternAndContext inferPredicatePatternAndContext(
       AspectPointcutCFlow pctCflow, String aspectFQN ) {
2    val _predicate = pctCflow.toALIA4JCFlowPredicate(
3      inferSpecialization(pctCflow.pattern,aspectFQN))
4    return new PredicatePatternAndContext(_predicate, null, new LinkedList
       ())
5  }
```

5.2 Security example using the AspectJ Compiler

This section describes the compilation of an example program using the AspectJ compiler that is discussed in the previous section. Assume that a file *Security.aj* contains an AspectJ program as in Listing 5.4.

Listing 5.4: Example AspectJ program

```
1 public aspect Security {
2
3   pointcut secureAccess()
4         : execution( * ajia.model.Account.update(..));
5
6   before() : secureAccess() {
7     System::out.println("Security Applied!");
8   }
9
10 }
```

Language models. When this program is compiled, an AspectJ model is generated that is depicted in figure 5.3.

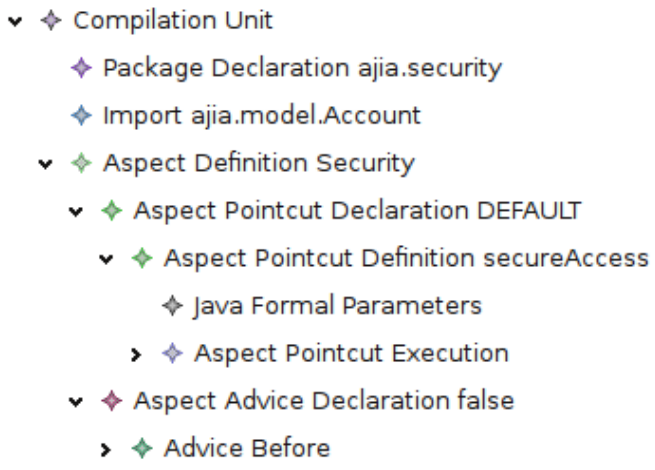

Figure 5.3: The AspectJ model resulting from the code in Listing 5.4

Model Inference. The language model is fed to the `AspectJJvmModelInferrer` that is used to infer an ALIA4J and a Jvm model. This Jvm model is specific to the Java code that is embedded in the advice part of the Security aspect. This ALIA4J model is

given to the `ALIA4JJvmModelInferrer` to infer a Jvm model. This is shown in Figure 5.2 as *ALIA4J Jvm model* and is specific to ALIA4J configurations.

Code Generation. After this process, a resource containing both models is given to the CodeGenerator. This is depicted in figure 5.4.

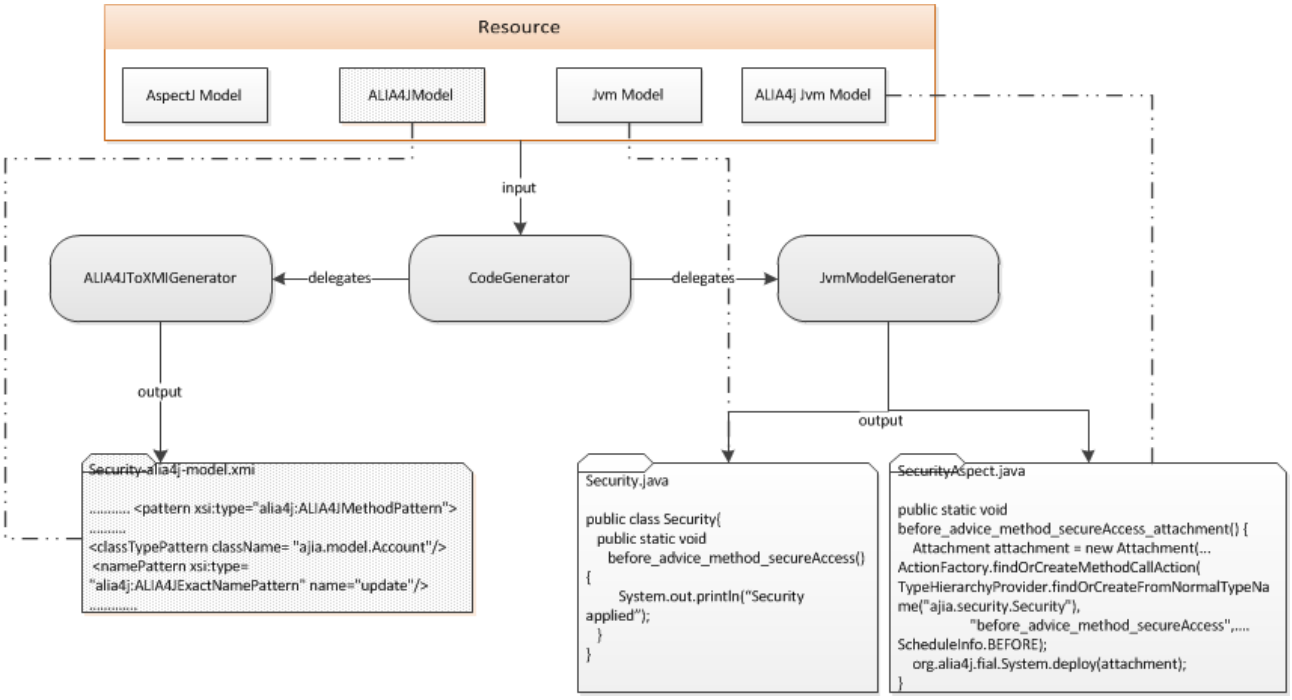

Figure 5.4: Generating code in the AspectJ Compiler

The CodeGenerator delegates to the `ALIA4JToXMIGenerator` to serialize the ALIA4J model to an XMI file called `Security-alia4j-model.xmi`. This XMI file can be viewed graphically using eclipse and is displayed in Figure 5.5.

The CodeGenerator also delegates to the `JvmModelGenerator` to generate Java code from the Jvm models. From the Jvm model, a Java file *Security.java* will be created with code that is shown in Listing 5.5. This class contains a method for each advice part of the aspect with the appropriate Java code from the aspect.

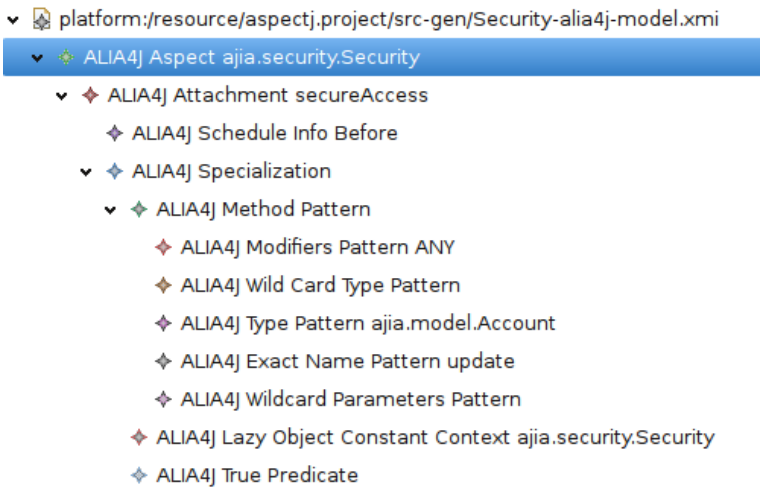

Figure 5.5: The serialized ALIA4J model as XMI

Listing 5.5: Translating Jvm model into Security.java

```
public class Security {
  public void before_advice_method_secureAccess() {
    System.out.println("Security Applied!");
  }
}
```

The ALIA4J Jvm model will be transformed into a Java file called *SecurityAspect.java*. It contains Java code for ALIA4J as shown in Listing 5.6.

Listing 5.6: Translating the ALIA4J Jvm model into SecurityAspect.java

```
public class SecurityAspect {
  private static Object secureAccess_specialization;

  public static void pointcut_secureAccess() throws ClassNotFoundException
      {


    MethodPattern pattern = new MethodPattern(
                        ModifiersPattern.ANY ,
                        TypePattern.ANY,
                        new ExactClassTypePattern(TypeHierarchyProvider.
                            findOrCreateFromNormalTypeName("ajia.model.
                            Account")),
                        new ExactNamePattern("update"),
                          ParametersPattern.ANY,
                          ExceptionsPattern.ANY);
```

```
    Predicate<AtomicPredicate> predicate = TruePredicate.<AtomicPredicate>
        truePredicate();

    secureAccess_specialization = new Specialization(pattern,predicate,
    Arrays.asList(
    ContextFactory.findOrCreateLazyObjectConstantContext(
        TypeHierarchyProvider.findOrCreateFromNormalTypeName("ajia.
        security.Security"))
    ));
  }

  public static void before_advice_method_secureAccess_attachment() {
    Attachment attachment = new Attachment(
      Collections.singleton((Specialization)secureAccess_specialization),
      ActionFactory.findOrCreateMethodCallAction(
        TypeHierarchyProvider.findOrCreateFromNormalTypeName("ajia.
            security.Security"),
        "before_advice_method_secureAccess",
        TypeHierarchyProvider.findOrCreateFromClasses(new Class[] {  }),
        TypeHierarchyProvider.findOrCreateFromClass(Void.class),
        ResolutionStrategy.VIRTUAL), ScheduleInfo.BEFORE);

    org.alia4j.fial.System.deploy(attachment);
  }

  public static void setupAT() throws ClassNotFoundException {
     ajia.security.SecurityAspect.pointcut_secureAccess();
     ajia.security.SecurityAspect.
         before_advice_method_secureAccess_attachment();
  }
}
```

5.3 Evaluation

In this section, an evaluation is given for the generic framework. During this evaluation the goals defined in section 2.3.4 are considered. These goals are Re-usability, Readability and Extensibility.

Re-usability is one of the goals stated in section 2.3.4. This goal has been reached by implementing the generic framework in a different eclipse project which is then used when implementing an aspect-oriented language. During the development of the AspectJ compiler, the language developer used the ALIA4J meta-model and helper classes from the framework to infer an ALIA4J model from the AspectJ model. The framework then used this ALIA4J model to generate Java code specific to ALIA4J.
When implementing a new aspect-oriented language, the generic framework can be used in the same way.

Readibility. Section 2.3.4 explained that code templates are in general difficult to understand. They are meta-programs that can produce code for multiple programs. They make compilers as a result less readable. One way to make the compiler implementation more readable is to hide the code template from the language developer.
This goal has been reached because the framework includes the code template to generate ALIA4J specific Java code. This is hidden from the language developer. This makes the language compiler much more readable then when the same compiler is implemented using Xtext but without our generic framework.
As an illustration, we developed an AspectJ compiler based upon Xtext before building the framework. The `AspectJJvmModelInferrer` used by this compiler is much more bloated then when using the framework. The reason for this is because in the implementation without the framework, code templates to generate Java code specific to ALIA4J are included. Including those code templates results in much more bloated code and effort from the language developer. This is not the case when using the framework because these code templates are already provided by the framework.

Another fact to point out is that the number of lines of code from the `AspectJJvmModelInferrer` decreased from 436 to 341. This is about 100 lines less code. Although these numbers may seem small, it must be pointed out that we only built a prototype of the AspectJ compiler.

Extensibility. In section 2.3.4 we defined extensibility as the support for the addition of possible LIAM refinements for new languages. ALIA4J provides an abstract language-independent meta-model with refinements for concrete languages. This might not be sufficient when implementing a new language. New refinements may be needed to express a construct of a new language.
There is support for the addition of possible LIAM refinements as described by section 4.3. Take as an example the implementation of the AspectJ compiler described in section 5.1. Before building the AspectJ compiler, the Ecore version of the ALIA4J meta-model did not have a predicate to describe the semantics of the `cflow` construct of AspectJ. A `CFlowPredicate` has been added to the ALIA4J meta-model and the `ALIA4JJvmModelInferrer` has been subclassed by a class called `CustomALIA4JJvmModelInferrer`. This allowed the AspectJ compiler to describe the semantics of `cflow` and generate code for it.
Other ways to improve the extensibility of the framework are described in section 4.3.

5.4 Related Work

The AspectBench Compiler (abc) [12] is a framework that supports the development of extensions to AspectJ. The abc compiler consists of a Polyglot and Soot block. Polyglot is used as a frontend to perform syntax and type checking for the language. Soot is a bytecode analysis toolkit and is used by abc to perform code generation. When developing an aspect-oriented language using abc, an existing language grammar can be modified and new passes through the abstract syntax tree (AST) can be added in Polyglot. Unlike the approach in our framework, the abc compiler allows advice code to be woven with Java classes. The AST of the aspect-oriented program is split into a pure Java AST and an AspectInfo structure that contains aspect-specific information. The AspectInfo is used for weaving the aspect-oriented program. In our approach aspect weaving is done by ALIA4J and the code generation is done by our framework. As opposed to abc, our generic framework can be used for different advanced dispatching languages and is not only restricted to languages with a syntax similar to AspectJ.

6

Conclusions & Future Work

6.1 Conclusion

The ALIA4J approach eases the burden of implementing advanced-dispatching languages by providing a language-independent advanced-dispatching meta-model (LIAM) to express semantics of language constructs. LIAM is implemented as a set of Java classes. ALIA4J also provides an execution environment that can be shared across language implementations.

The compilation to Java code that uses the ALIA4J API is not trivial and code templates that generate this API code can become complex and bloated. Coding the translation to ALIA4J specific Java code has to be performed by the language developer for every language implementation when implementing compilers using this approach.

This thesis proposed a generic framework that can be re-used to build aspect-oriented language compilers. ALIA4J specific Java code is generated by the framework by including code templates for this Java code. The generic framework is re-usable, leads to more readable compiler implementations, and is extensible by supporting the addition of new LIAM entity refinements.

The framework is built upon Xtext. Xtext supports the development of programming languages by generating components such as a lexer and parser. It also generates an eclipse-based IDE that is tailored for the language by providing code highlighting and content assist. When a program is parsed, an Ecore model is created that represents the structure of the program. We call this model a *language model*.

We developed an *ALIA4J meta-model*, which is an Ecore version of LIAM that is usable in Xtext. This meta-model is included with the framework. The framework allows an ALIA4J model and a Java model to be inferred from the language model.

These models are then used to generated code.

This thesis also presents an AspectJ compiler that is built using the framework. The implementation demonstrates the use of the framework without considering ALIA4J API calls in Java.

6.2 Future Work

A discussion is given regarding the extensibility of the framework. This is an important aspect since new languages may have constructs that cannot be expressed with existing LIAM entities. The thesis draws the conclusion that the extensibility of the current framework can be increased. Solutions are discussed that can be used to make the framework more extensible in the future.

Another area to focus on during future work is the possible complexity of boolean expressions in pointcut expressions. These expressions will be mapped to different ALIA4J entities such as Specializations, Patterns and Predicates. ALIA4J only allows the negated normal form of Predicate expressions. This makes it complex since the pointcut expression first has to be transformed into the negated normal form. This complexity is also shared among compilers since multiple languages allow conjunctions and disjunctions in their pointcut expressions. This kind of complexity could be hidden from the language developer by allowing the framework to do this transformation.

A

ALIA4J Generic Framework

The code in Listing A.1 shows part of the `ALIA4JModelInferrer`. It includes code templates for the body of the methods that will be generated.

Listing A.1: Part of the `ALIA4JModelInferrer` with code templates

```
1
2   def dispatch generateALIA4JPattern(ALIA4JMethodPattern pattern, String
        varName) '''
3     «IF pattern.parametersPattern instanceof ALIA4JExactParametersPattern»
4       TypeDescriptor[] types = new TypeDescriptor[«(pattern.
            parametersPattern as ALIA4JExactParametersPattern).
            typeDescriptors.size»];
5       «var i = -1»
6       «FOR type: (pattern.parametersPattern as
            ALIA4JExactParametersPattern).typeDescriptors»
7         types[«i = i+1»] = TypeHierarchyProvider.findOrCreateFromClass(
              Class.forName("«type»"));
8       «ENDFOR»
9     «ENDIF»
10
11    MethodPattern «varName» = new MethodPattern(
12    «pattern.modifiersPattern.generateALIA4JModifiersPattern»,
13    «pattern.typePattern.generateALIA4JTypePattern»
14    «pattern.classTypePattern.generateALIA4JClassTypePattern»
15    «pattern.namePattern.generateALIA4JNamePattern»,
16    «pattern.parametersPattern.generateALIA4JParametersPattern»
17    ExceptionsPattern.ANY);
18
19  '''
20
21  def dispatch generateALIA4JPattern(ALIA4JFieldPattern pattern, String
        varName) {
22
23    var write = false
```

```
24      switch pattern {
25        ALIA4JFieldWritePattern : {
26          write = true
27        }
28        ALIA4JFieldReadPattern : {
29          write = false
30        }
31      }
32      val _write = write
33      '''
34        «IF _write»FieldWritePattern «varName» = new FieldWritePattern(
35        «ELSE»FieldReadPattern «varName» = new FieldReadPattern(
36        «ENDIF»
37        ModifiersPattern.«pattern.modifiersPattern.visibility.toString» ,
38        «pattern.typePattern.generateALIA4JTypePattern»
39        ClassTypePattern.ANY,
40        «pattern.namePattern.generateALIA4JNamePattern»);
41      '''
42    }
43    def dispatch generateALIA4JPredicate(ALIA4JTruePredicate p, String
          varName) '''
44      Predicate<AtomicPredicate> «varName» = TruePredicate.<AtomicPredicate>
            truePredicate();
45    '''
46
47    def dispatch generateALIA4JPredicate(ALIA4JInstanceOfPredicate p, String
           varName) '''
48      Predicate<AtomicPredicate> «varName» = new BasicPredicate<
            AtomicPredicate>(AtomicPredicateFactory.
            findOrCreateInstanceofPredicate(
49                      «p.context.generateALIA4JContext»,
50                        TypeHierarchyProvider.findOrCreateFromNormalTypeName
                              ("«(p as ALIA4JInstanceOfPredicate).type»")),
                              true);
51    '''
52
53    def dispatch generateALIA4JContext(ALIA4JContext context) '''
54        «IF context instanceof ALIA4JCalleeContext» ContextFactory.
              findOrCreateCalleeContext()
55        «ELSEIF context instanceof ALIA4JCallerContext» ContextFactory.
              findOrCreateCallerContext()
56        «ELSEIF context instanceof ALIA4JLazyObjectConstantContext»
              ContextFactory.findOrCreateLazyObjectConstantContext(
              TypeHierarchyProvider.findOrCreateFromNormalTypeName("«(context
              as ALIA4JLazyObjectConstantContext).type»"))
57        «ELSEIF context instanceof ALIA4JArgumentContext» ContextFactory.
              findOrCreateArgumentContext(«(context as ALIA4JArgumentContext).
              argument»)«ENDIF»'''
```

Bibliography

[1] Christoph Bockisch, Andreas Sewe and Martin Zandberg: ALIA4J's [(Just-In-Time) Compile-Time] MOP for Advanced Dispatching

[2] www.xtext.org

[3] Xtext Reference Documentation at http://www.eclipse.org/Xtext/documentation/2_1_0/Xtext%202.1%20Documentation.pdf

[4] Xtend Reference Documentation at http://www.eclipse.org/xtend/documentation/index.html

[5] JSR-330, http://jcp.org/aboutJava/communityprocess/final/jsr330/index.html

[6] Google Guice, http://code.google.com/p/google-guice/

[7] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[8] C. Bockisch, S. Malakuti, M. Aksit, and S. Katz. Making aspects natural: Events and composition. In Proceedings of AOSD, 2011.

[9] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. TOSEM, 20(1), 2010.

[10] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. Escala: Modular event-driven object interactions in scala. In AOSD' 11, 2011

[11] C. A. Pavel, C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. D. Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In OOPSLA, pages 345-364, 2005.

[12] AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L. J., KUZINS, S., LHOTAK, J., LHOTAK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2006. abc: An extensible AspectJ compiler. Trans. Aspect-Orient. Softw. Develop. 1, 293-334.