Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming

Parisa Mirshams Shahshahani

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Software Engineering Concordia University
Montreal, Quebec, Canada
April 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Parisa Mirshams Shahshahani

Entitled: Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Software Engineering

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. N. Shiri

_____ Examiner
Dr. Yuhong Yan

_____ Examiner
Dr. Joey Paquet

_____ Supervisor
Dr. Constantinos Constantinides

_____ Co-supervisor
Dr. Abdelwahab Hamou-Lhadj

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

Abstract

Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming

Parisa Mirshams Shahshahani

Software engineers often have to resort to various program analysis tools to analyze the structure and sometimes behavior of a system before they can make changes that preserve system reliability and other quality attributes. The Knowledge Discovery Metamodel (KDM) is an OMG standard which specifies a language-independent representation of the programs to be analyzed. The advantages of using KDM are numerous including an increase in productivity and a cut in overall costs during maintenance, as it allows for a reuse of available KDM compatible tools and expertise. Currently, KDM supports a number of procedural and object-oriented programming languages. However, no support currently exists for aspect-oriented programming languages. This thesis aims at filling this gap, by extending KDM to support AspectJ, perhaps the most popular aspect-oriented language. We show an application of the extended model to an aspect-oriented application.

Acknowledgments

This thesis would not have been possible without the help, support and patience of my supervisor, Dr. Constantinos Constantinides whose advice, encouragement and unsurpassed knowledge contributes to my graduate work and experience. I would have been lost without him.

I would like to express my deep gratitude to my co-supervisor Dr. Abdelwahab Hamou-lhadj whose good advice and support has been invaluable on this research. I warmly thank my best friend and my best colleague, Zohreh Sharafi for all the emotional support, camaraderie and encouragements. I also would like to thank the members of Software Maintenance and Evolution Research Group, Michel Parisien and Saeed Bohlooli for providing valuable comments and their friendly help for my research work.

Lastly, and most importantly, I owe my loving and sincere thanks to the most influential people in my life: my parents, Maryam Azimi and Mostafa Mirshams for unconditional support and encouragement to pursue my interest.

Contents

List of Figures

Chapter 1

Introduction and Motivation

In order to gain comprehension of source code, *program analysis* is a collective term used to refer to various methods and techniques, normally supported by automation and tools, deployed to obtain knowledge of the structure or the behavior of source code, thus distinguishing between static and dynamic program analysis. Currently, a wide variety of tools exist which read the structure (source code) or the behavior (execution traces) of programs, each one exhibiting its own strengths and weaknesses. However, most tools tend to be language specific, and as a result maintainers sometimes cannot combine them to utilize automation and tool support and achieve their maximum benefits.

The Knowledge Discovery Metamodel (KDM) [36] is a standard by the Object Management Group (OMG) [41] to support Architecture-Driven Modernization (ADM) [15]. This standard provides a specification at a level of abstraction higher

than source code in order to specify a language independent representation of programs. The KDM currently supports a wide collection of programming languages [36].

The KDM is an intermediate representation of software artifacts and its operational environment. The KDM specification can be used as a language and platform independent representation for analyzing software systems. This allows the incremental analysis of a software system, where each tool reads and analyzes the KDM representation, extracts precise knowledge and if required adds more knowledge to it. The ultimate benefit of KDM is to provide interoperability among different tools for various maintenance and evolution tasks. It enhances the maintenance of a software system due to the existence of a uniform representation of programs that can be read by all KDM-compatible analysis tools. This way, we do not need to provide a point-to-point transformation of program representations to allow the compatibility of analysis tools. As a consequence, we can use several individual analysis tools where each one provides one specific type of analysis that maintainers would need at a time. This allows the reuse of available technologies and expertise while reducing costs associated with maintenance which is reported to consume a high proportion of the overall costs during the software life cycle.

An alternative to KDM is to rely on abstract syntax trees (AST) to represent the source code. Although both AST and KDM are middle representations an AST model differs from one programming language to another, whereas the KDM forms a

common representation for every programming language it supports, enabling sharing and reusing of data among analysis tools.

Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties in object-oriented systems cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localized in single modular units [16]. Their implementation ends up cutting across the decomposition of the system. Examples of these crosscutting concerns (or aspects) include persistence, authentication, synchronization and contract checking. Aspect-Oriented Programming (AOP) [24] explicitly addresses those concerns by introducing the notion of an aspect, which is a modular unit of decomposition. Currently there exist many approaches and technologies to support AOP. One notable technology is AspectJ [28], a general-purpose aspect-oriented language, which has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a common vocabulary based on its own linguistic constructs. In the AspectJ model, an aspect definition is a unit of modularity providing behavior to be inserted over functional components. This behavior is defined in method-like blocks called an advice. A pointcut expression is a predicate over well-defined points in the execution of the program called join points. Even though the specification and the level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to and execution of methods and constructors. When the

program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated join point. Furthermore, several advice blocks may apply to the same pointcut. The order of execution can be specified by rules of advice precedence specified by the underlying language [29].

The KDM supports a number of procedural and object-oriented programming languages (like C++ and Java). However, no support currently exists (or has been proposed) for the AspectJ programming language; currently the most popular general-purpose aspect-oriented language with an increasing community of researchers and practitioners in academia and industry. The objective of this thesis is to extend the KDM metamodel in order to provide support for AspectJ.

1.1 Thesis Outline

The remainder of this thesis is organized as follows: In Chapter 2 we provide the necessary background to Knowledge Discovery Metamodel (KDM), aspect-oriented programming (AOP), model transformations, and at the end we discuss related work. In Chapter 3 we discuss our overall approach. Later in this Chapter we describe our methodology: how we integrate AspectJ constructs into the KDM metamodel by

introducing the AspectKDM metamodel. Additionally, we describe a model-to-model transformation for mapping aspect models. In Chapter 4 we describe a case study to demonstrate how the proposed metamodels can be applied in an AspectJ project. We present our conclusions and recommendations in Chapter 5.

Chapter 2

Background

2.1 Knowledge Discovery Metamodel (KDM)

The Knowledge Discovery Metamodel (KDM) is a standard defined by the Architecture-Driven Modernization (ADM) Task Force in OMG. The KDM is defined via the Meta-Object Facility (MOF) [40], a standard by OMG to write metamodels, and uses XML Metadata Interchange (XMI) [35] model interchange format, an OMG standard for exchanging metadata information via Extensible Markup Language (XML). The KDM defines its KDM XMI schema, that each KDM instance should conform to. The KDM is a metamodel for representing information about an existing software system, its elements and its operational environment. The KDM specification [36] groups the information about an existing system into different architectural view points called *domains*. Each domain corresponds to one KDM model and one package, with the

exception of `Code model` which is split between `Code` and `Action` packages. KDM consists of four different layers of abstraction, each level is based on the previous one [36].

The KDM layers and their packages are represented in Figure 1. Figure 2 illustrates the layers and the dependencies between KDM packages in different KDM layers.

1. **Infrastructure Layer:** It is the lowest abstraction level, which consists of a small set of common metamodel elements (such as entity and relationship) used through entire KDM levels. This layer consists of three packages: `Core`, `KDM` and `Source`.

2. **Program Elements Layer:** It represents the code elements and their associations. It consists of a broad set of metamodel elements common between different programming languages to provide a language-independent representation. There are two packages in this layer: `Code` and `Action`. They both use the `Code model`.

3. **Runtime Resource Layer:** It represents higher-level knowledge (such as operational environment) about existing software systems. This kind of knowledge cannot be extracted from the syntax at code level but rather from the runtime incremental analysis of the system. There are four packages in this layer: `Data`, `Event`, `UI` and `Platform`.

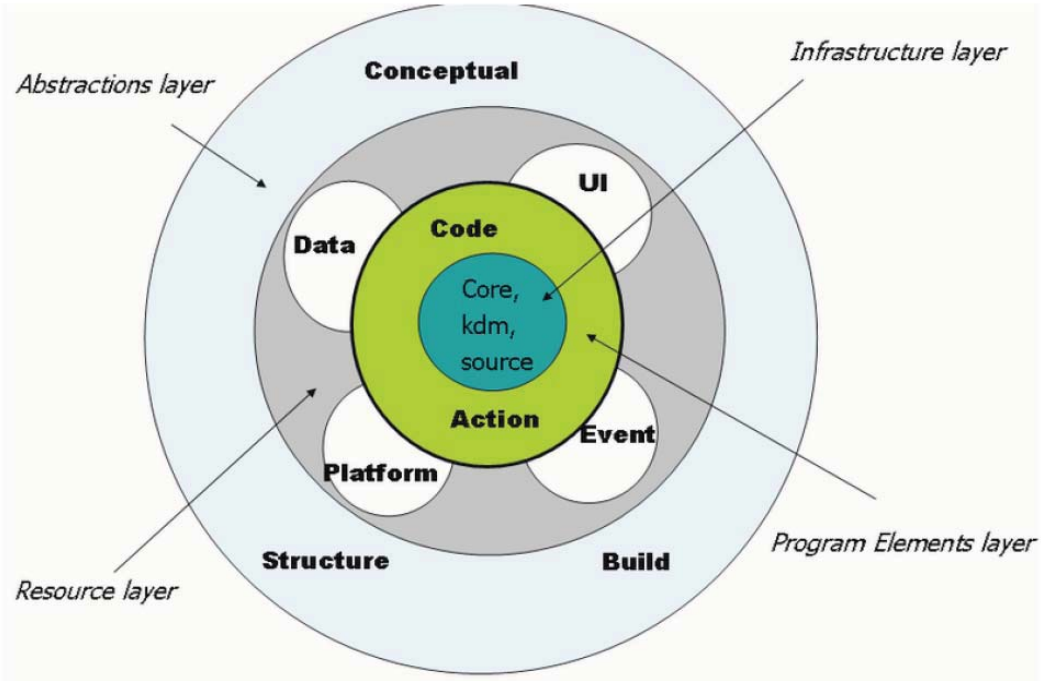

Figure 1: Knowledge Discovery Metamodel layers, packages, and separation of concerns. (Adopted from the KDM specification [36])

4. **Abstractions Layer:** It defines a set of metamodel elements for representing domain and business-specific overview of the system. Extracting this kind of knowledge involves input from experts and analysts. `Conceptual`, `Structure` and `Build` are the three packages in this layer.

2.2 Aspect-Oriented Programming (AOP)

In 1974, Edsger W. Dijkstra in his paper "On the role of scientific thought" [14] discussed Separation of Concerns (SoC). It states that a complex system has different kinds of concerns that should be identified and treated separately. Using the SoC

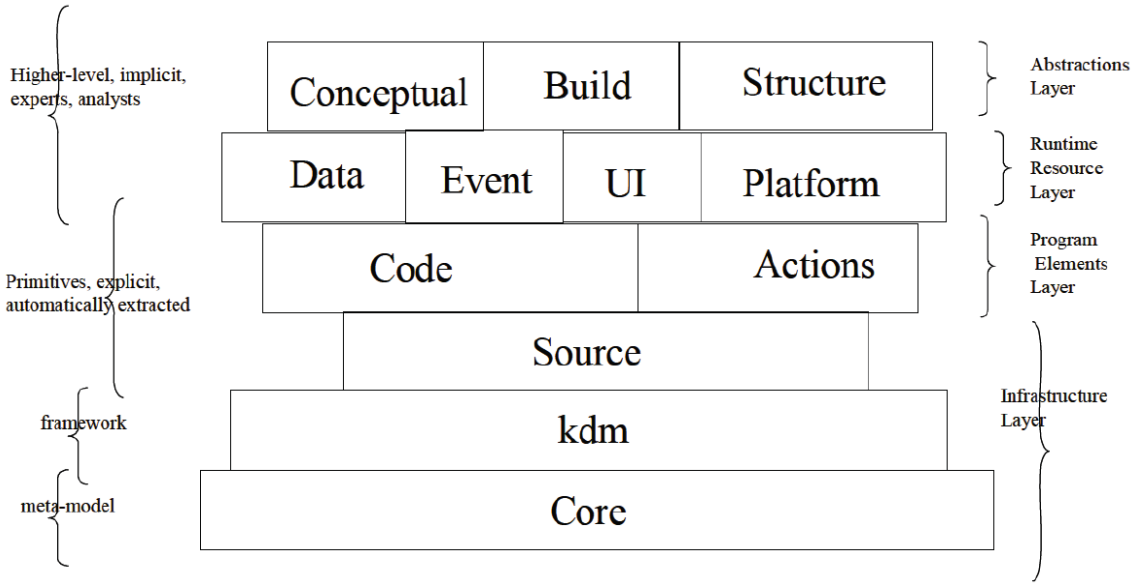

Figure 2: Structure of KDM packages. (Adopted from the KDM specification [36])

can lead to better quality factors such as robustness, adabtability, maintainability, and reusability [12]. Various programming paradims allow us to achive separation of concerns at the code level. Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties in object-oriented systems cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localized in single modular units [16]. Their implementation ends up cutting across the decomposition of the system. Figure 3 represents an initial picture of the crosscutting concerns. Persistence, authentication, synchronization and contract checking are all examples of crosscutting concerns. Crosscutting imposes two symptoms on software development which are represented in Figure 4:

1. Code scattering: Implementation of some concerns not well modularized but

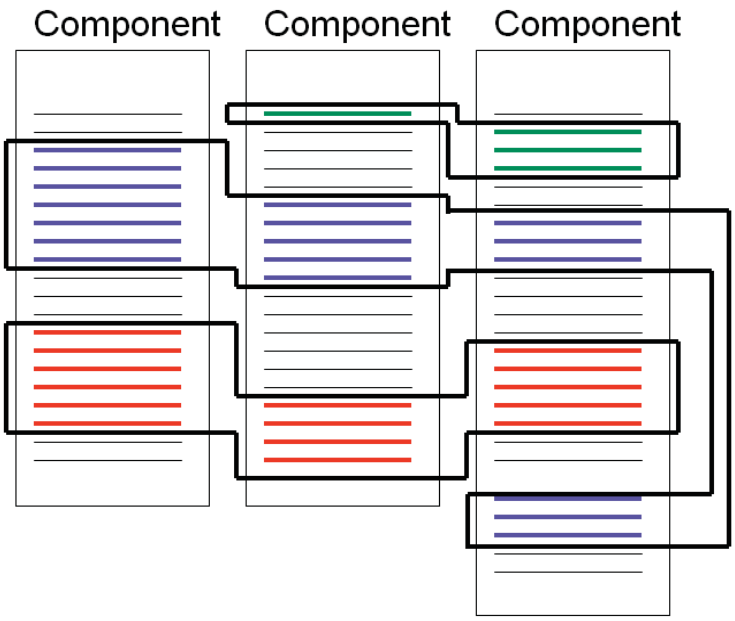

Figure 3: Croscutting concerns can affect system components both horizontally as well as vertically.

cuts across the decomposition hierarchy of the system.

2. Code tangling: A module may contain implementation elements (code) for various concerns.

Aspect-oriented Programming (AOP) [24] provides a new unit of decomposition (aspects), that explicitly captures the crosscutting concerns (see Figure 5). Currently there exist many approaches and technologies to support AOP. One notable technology is AspectJ [28], a general-purpose aspect-oriented language, which has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a common vocabulary based on its own

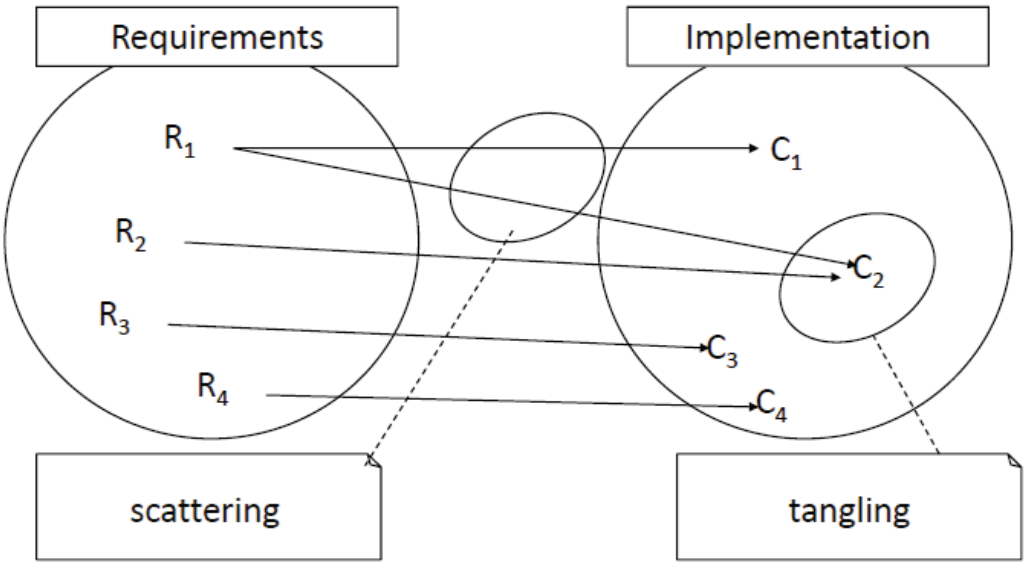

Figure 4: Illustration of scattering and tangling as two symptoms of crosscutting.

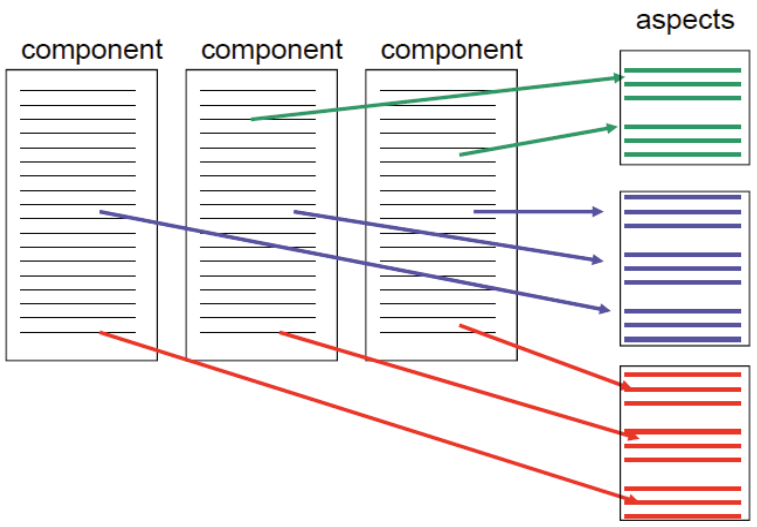

Figure 5: Initial picture of separation of concerns. Ideally one would want to move from the picture of Figure 3 to the one shown here, where a compelete separation of concerns is achieved.

linguistic constructs. In the AspectJ model, an aspect definition is a unit of modularity providing behavior to be inserted over functional components. This behavior is defined in method-like blocks called an *advice*. A *pointcut* expression is a predicate over well-defined points in the execution of the program called *join points*. Even though the specification and the level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to and execution of methods and constructors. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated join point. Furthermore, several advice blocks may apply to the same pointcut. The order of execution can be specified by rules of advice precedence specified by the underlying language [29].

2.2.1 The Aspect Bench Compiler for AspectJ (abc)

The Aspect Bench Compiler for AspectJ (abc) is an extensible AspectJ Compiler which is freely available under the GNU LGPL. The abc compiler is designed to provide a workbench for aspect-oriented programming research and investigation that facilitates adding new features to the AspectJ language. In addition, the abc compiler has suggested a grammar for AspectJ [2] as an extension to Java grammar. The compiler is based on *Polyglot* [34] as its frontend and *Soot* [51] as its backend. Figure

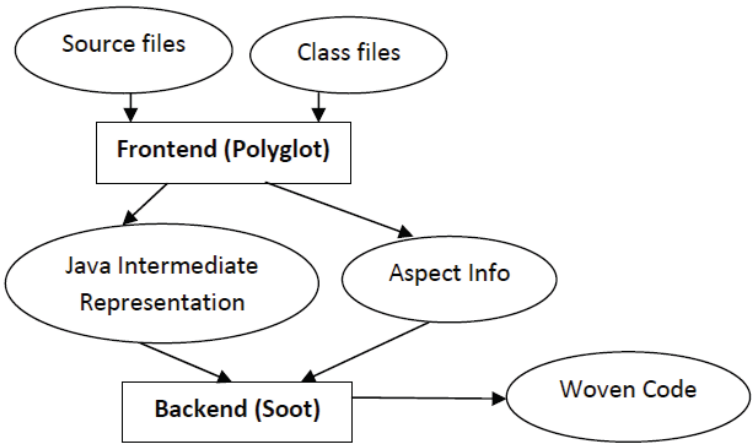

Figure 6: The abc compiler simple design. (Adopted from [1])

6 illustrates a simple design for the abc compiler.

Figure 7 illustrates the abc compiler architecture. Polyglot is an extensible compiler framework for Java. It uses Java as the base language and adds the extensions as a collection of separate files to it. The Polyglot frontend parses AspectJ source code into an abstract syntax tree (AST) and then runs a number of passes to perform type checking and to separate AspectJ specific construct information. The output of this phase is a pure Java AST and the AspectJ constructs called Aspect Info. The Java AST is the AspectJ program with the AspectJ constructs completely removed, but it has some placeholders for these constructs to be filled in the weaving phase by the Soot backend.

Soot is a framework for analyzing and transforming Java bytecode. Soot uses an intermediate representation called *Jimple* which is a typed, stack-less, three address code. Deploying Soot enables the abc compiler to provide optimization of AspectJ

code after weaving is completed. In the abc compiler, Polyglot is used as the frontend and provides Soot with the AST of the program. Then Soot translates the AST into Jimple intermediate representation (IR). After processing, the abc compiler uses the Jimple IR and the Aspect Info construct and weave them to make bytecode. More information on the abc compiler can be found in [5].

2.3 Model Transformations

Model Driven Engineering (MDE) refers to a range of development approaches that are based on the use of software modeling as a primary form of expression. Model transformations plays an important role in MDE, since models are the main development artifacts which drive the process of MDE. A model transformation is an automated process that maps a source model to a target model according to a set of rules. Both source and target metamodels conform to specific metamodels. Figure 8 illustrates the four-layer metamodel architecture by OMG. In the four-layer metamodel architecture, `M3` represents the metametamodel layer which defines a small set of concepts for defining and manipulating the models of metamodels. This allows new metamodels and modeling languages to be defined. The Object management Group (OMG) proposes Meta Object Facility (MOF) to be the standard metametamodel. `M2` is the metamodel layer, which defines the structure, semantics, and constraints for a family of models. In other words, it specifies the concepts of the language used to

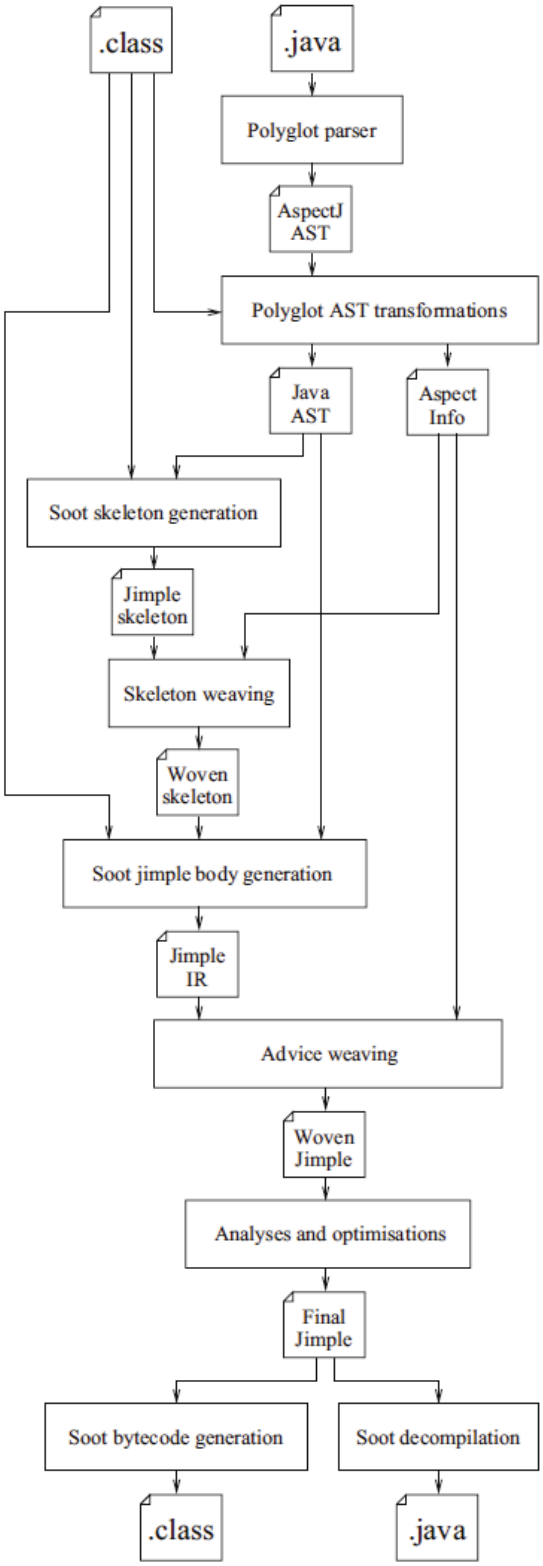

Figure 7: The abc compiler architecture. (Adopted from [5])

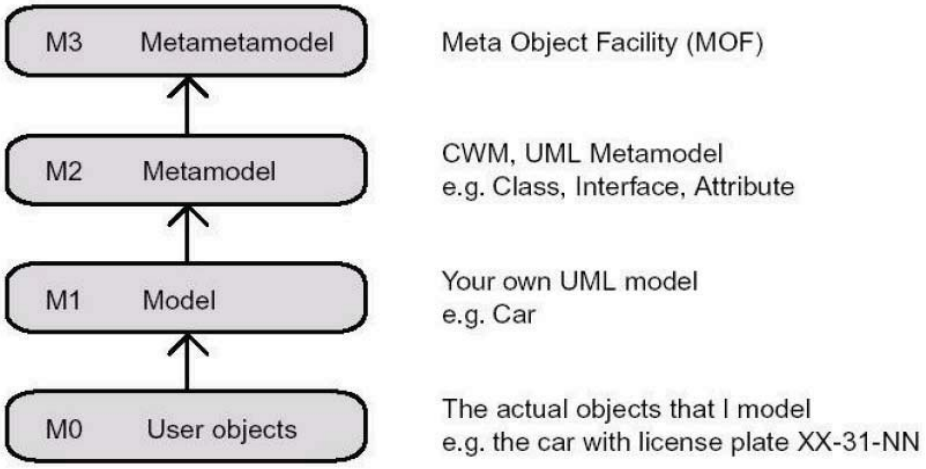

Figure 8: The four-layer metamodel architecture.

define a model. An example of the metamodel is the UML metamodel. Metamodels that are defined using the same metametamodel can exchange information. `M1` is the model layer and finally `M0` is the object layer. The relation between the models expressed in a language and the metamodel of that language is called `conformsTo` (model conforms to a certain metamodel) [31].

The Model Transformation pattern as shown in Figure 9 is a common pattern followed by model transformations in MDE: the `Tab` is the transformation program which needs `Ma` as the source model and provides `Mb` as the target model. The `Ma`, `Mb` and `Tab` are all models conforming to `MMa`, `MMb`, and `MMt` metamodels respectively. The `MMM` is the corresponding metametamodel for all of the metamodels `MMa`, `MMb` and `MMt`. In the context of OMG, `MMM` is the MOF.

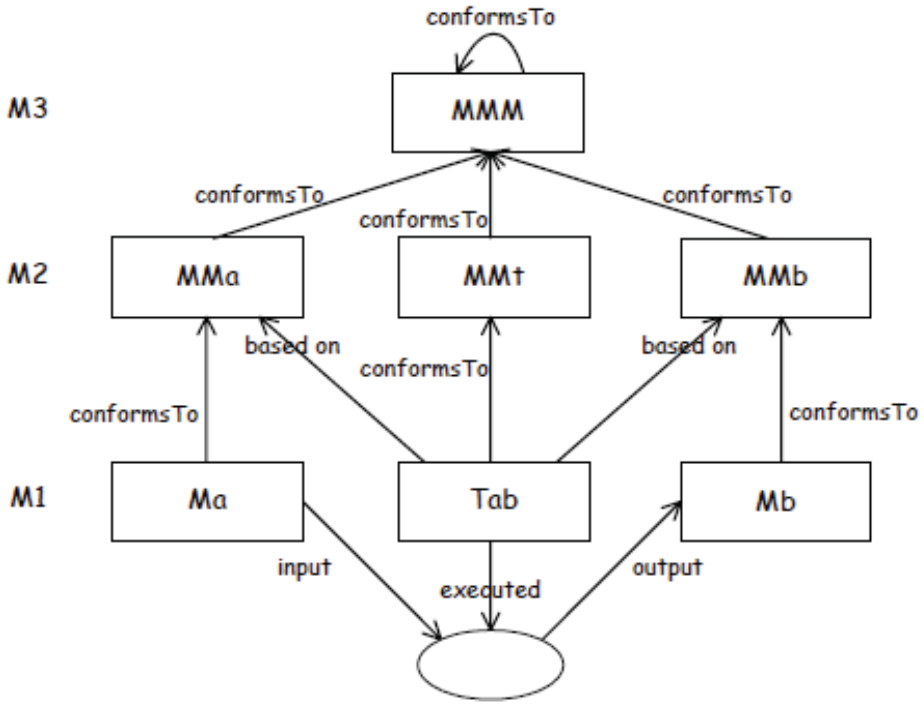

Figure 9: The model transformation pattern. (Adopted from [25])

2.3.1 The ATLAS Transformation Language (ATL)

The ATLAS Transformation Language (ATL) is a transformation language developed as a part of the AMMA (ATLAS Model Management Architecture) platform [25]. ATLAS deploys the model transformation pattern illustrated in Figure 10. In this pattern `mma2mmb.atl` is the transformation definition written in the ATL language to automatically transform `Ma` model to `Mb` model. The transformation definition `mma2mmb.atl` is a model which conforms to the ATL metamodel. The `MMa`, `MMb` and ATL all conform to the MOF.

The ATL is inspired by Query/View/Transformation (QVT) by OMG [39]. In

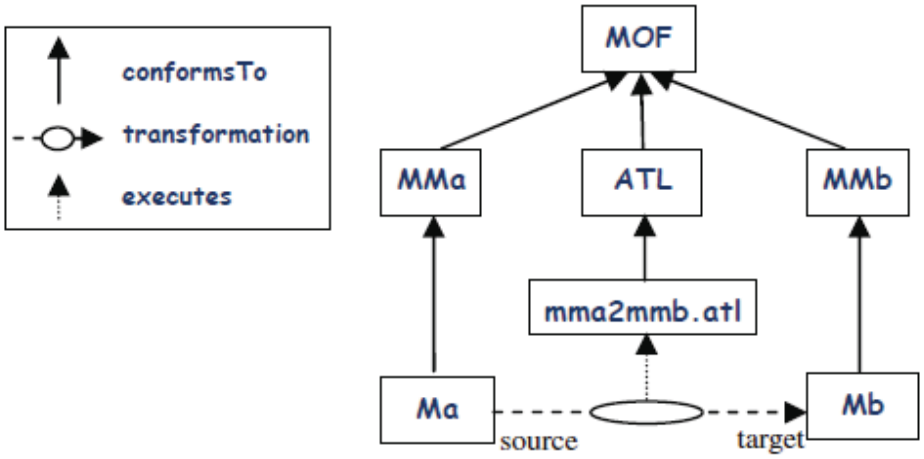

Figure 10: Overview of the ATL transformational approach. (Adopted from [26])

fact, this is the reason why ATL is compatible with Model Driven Architecture (MDA) standards like UML, OCL [37] and MOF. The ATL metamodel is compatible with XML Metadata Interchange (XMI) [35] and therefore the source and the target metamodels can be presented in XMI format. ATL is able to perform all QVT transformation scenarios where the transformation definitions are based on MOF, and it is applicable in the transformation scenarios beyond the QVT context [27].

The ATL is a hybrid transformation language which provides both declarative and imperative language constructs. The declarative style is encouraged [25] because it is closer to the way programmers think so it hides complex tasks such as selecting the source and target elements, rule triggering, and defining traceability links behind a simple syntax. On the other hand, some of the transformation problems can be solved by imperative programming. Based on the problem at hand, a programmer

has the option to choose between declarative and imperative styles.

The ATL transformations are unidirectional. To initialize a bidirectional rule, two individual transformations must be implemented, one transformation for each direction.

An ATL transformation is defined by modules. The module is composed of the following three elements:

1. header
2. helper
3. rule

The header section defines the name of the module (which should correspond to a file name) and the source and target metamodels.

The term helper comes from the Object Constraint Language (OCL) specification [37]. The ATL helpers can be considered as equivalent to the Java methods, since they are used to avoid code redundancy and they can be called from any point in the ATL transformation. In ATL the helpers can be specified only on `OCL types` and source metamodel element.

The rules express the transformation logic in ATL. The ATL rules describe how an output element is generated from an input element. The ATL rules can be declarative or imperative. The ATL can have the following rules:

- Matched rule : declarative.

- Called rule : imperative.

The matched rule is the core of an ATL declarative transformation. They are the answers to the two following questions:

- For which kinds of source elements, target elements must be generated.
- How the generated target elements have to be initialized.

Note that a source element should not be matched in more than one transformation rule.

We have chosen ATL as the transformation language in this thesis for the following reasons:

- It is a standardized language that enjoys wide acceptance.
- There are some tools that support this language, currently the ATL Integrated Development Environment (IDE) is built on top of the Eclipse platform. It provides syntax highlighting, error reporting and debugging features as well.
- It can provide complete model transformations while its expressions are based on OCL and are not complex.

2.4 Related Work

2.4.1 Knowledge Discovery Metamodel

Dehlen *et al.* [13] propose an extension for KDM to model the interoperability between legacy systems. With the help of Interoperability Knowledge Discovery Metamodel (IKDM), they show how the Model-Driven Engineering (MDE) approach can be used to provide solution in interoperability context, and how to provide modeling for different interoperability patterns.

Gerber *et al.* [21] discuss the aspects of the legacy system that should be modeled to interest both the modeling community and the people involved in modernization of the legacy systems. They discuss the system attributes and key aspect that should include in Knowledge Discovery Metamodel for Architecture-Driven Modernization (ADM). They conclude that by using Model-Driven Architecture (MDA) approach it is possible to evolve legacy systems (with the use of QVT to abstract KDM models to platform-specific models (PSM) and platform-independent models (PIM)).

Moyer [33] discusses the legacy systems and introduces KDM as a software modernization solution. He then describes what KDM is and discusses the different layers of KDM and whether they can be made automatically or not. For example, existing tools can build the moel for Infrastructure and Program Element layers by scanning the code, but not for the Resource and Abstractions layers. Ulrich [50] also discusses the importance of modernization for legacy system and the need for a standards for

the modernization process. He then introduces KDM as a modernization standard, a part of OMG's ADM task force, that can be used as a common metamodel to exchange information independent to language and platform across tools for analyzing and refactoring the existing systems and transforming them into new ones.

The Model-Driven Engineering (MDE) goals, solutions, and application scope are discussed by Bezivin *et al.* [9]. The authors suggest that the KDM standard should be used when we need to extract more specific information from a system along with the UML diagrams.

Castillo *et al.* propose a modernization approach for recovering business processes from legacy systems (MARBLE) [43, 42]. MARBLE has four layers of abstraction. They have used KDM metamodel as their platform independent model (PIM) in L2 level, therefore the analytic tools can use this model and generate new knowledge and add it to this model. Finally they deploy QVT to transform the KDM models into the business process models. In [44], Castillo *et al.* claim that using MDD and specially KDM is one of the main highlights of their work and make it stand out from the other works done to recover the business processes.

2.4.2 Modeling Crosscutting Concerns

Reina *et al.* [45] provide an overview of some works for modeling aspects in UML. In this survey they discuss the contributions of each work, the level of abstraction and the language the metamodel is proposed for.

Before UML 2.0, the extension mechanism in UML was not integrated with metamodels, many of the extensions were partially text based, and the base metamodel and the extension were not fully separated. For example in [8] they deploy text based format for pointcuts. In [19] and [48] the extension is not fully separated from the base metamodel.

To extend UML 2.0 there are two options: heavyweight and lightweight extension mechanism. Heavyweight extension is introducing new meta-classes for UML. This requires new tool support for the proposed metamodel, thus it cost a lot. [22] is an example of heavyweight extension to UML. Building a UML profile [23] is the lightweight extension mechanism which defines stereotypes and tagged values for a subset of the current UML metamodel elements. This way modeling with the new profile can be done through well-known UML tools and developers are familiar with the notation and the concepts which reduces the costs. [14] and [7] are examples of proposing a UML profile.

In this work, we deploy the CoreAOP profile proposed by Sharafi *et al.* [46] as a base metamodel for AspectJ representation. The most important characteristic of the proposed UML profile is the language independence property, which can be used to provide metamodels for other aspect-oriented languages. Moreover, our proposed domain model for AspectJ is partially similar to Everman's [17] proposed AspectJ profile for UML. Our approach has been different, since in KDM the language constructs are more general to able other like languages modeled with the use of same

constructs. Therefore, our proposed metamodel for KDM can be extended easily in order to support any additional feature. Everman models join points by defining different kinds of pointcuts. In contrast, we model join point constructs separately from pointcuts.

Chapter 3

Methodology

In this chapter we first discuss our overall approach. Then we discuss our AspectInfo metamodel. Later in this Chapter we disscuss how we integrate AspectJ constructs into the KDM metamodel by introducing the AspectKDM metamodel. Additionally, we describe a model-to-model transformation for mapping aspect models.

3.1 Overall Approach

There is currently no Aspect Oriented support in KDM. Our objective is to fill this gap. In this Section, we discuss our research proposal.

Figure 11 represents the big picture for creating KDM representation from an AspectJ project. Figure 12 is the activity diagram for the steps we have to follow to create KDM representation. An AspectJ project contains Java and AspectJ files.

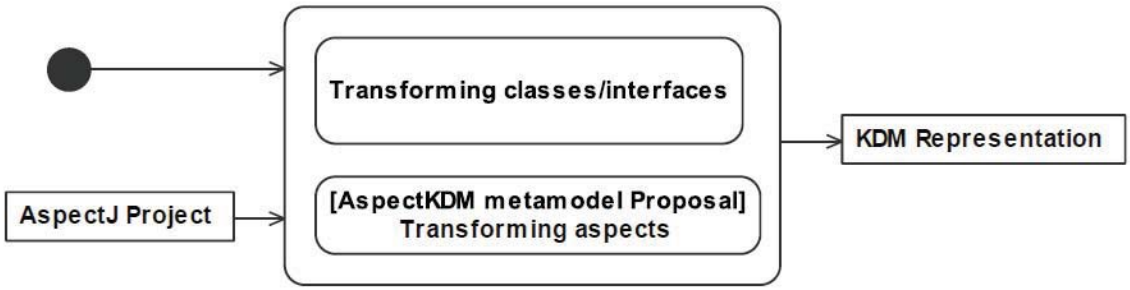

Figure 11: Creating the KDM representation from an AspectJ project (the big picture).

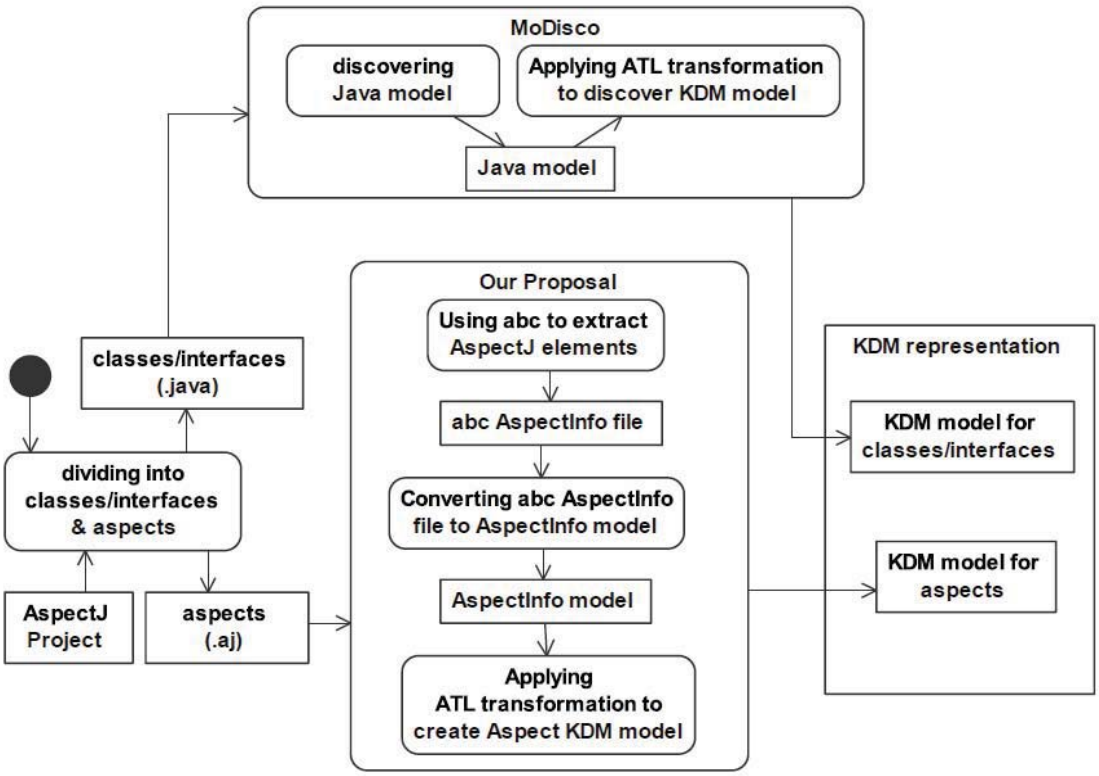

Figure 12: Creating the KDM representation from an AspectJ project.

We use `MoDisco` [6], [32] to discover the KDM representation for the class representation. `MoDisco` is an extensible open source project [10] for reverse engineering legacy systems. `MoDisco` is a MDE framework so it gives different kind of models (such as KDM representation) as the result of reverse engineering. To create a `KDM model` for AspectJ files we include the following steps:

1. We deploy the abc compiler to extract the information on crosscutting concerns. The abc compiler is provided as a Java stand-alone project and can be downloaded from the abc group website [2].

2. Converting the information we extract from AspectJ files into a model which conforms to the `AspectInfo` metamodel. Producing the `AspectInfo` metamodel is done using Eclipse Modeling Framework (EMF) [11]. In the previous section, we mentioned how a metamodel represented in EMF can be instantiated to create an instance model.

3. We apply ATL transformation on the `AspectInfo` model to generate `AspectKDM` representation. Using EMF, we provide the `AspectKDM` metamodel. Additionally we use ATL Eclipse plug-in for writing and execution ATL transformations.

The proposed procedure (see Figure 12) can be performed automatically using two Eclipse plug-ins (ATL and EMF) together with the abc compiler java project. Along with the MoDisco tool we can provide a complete KDM representation for an AspectJ project automatically.

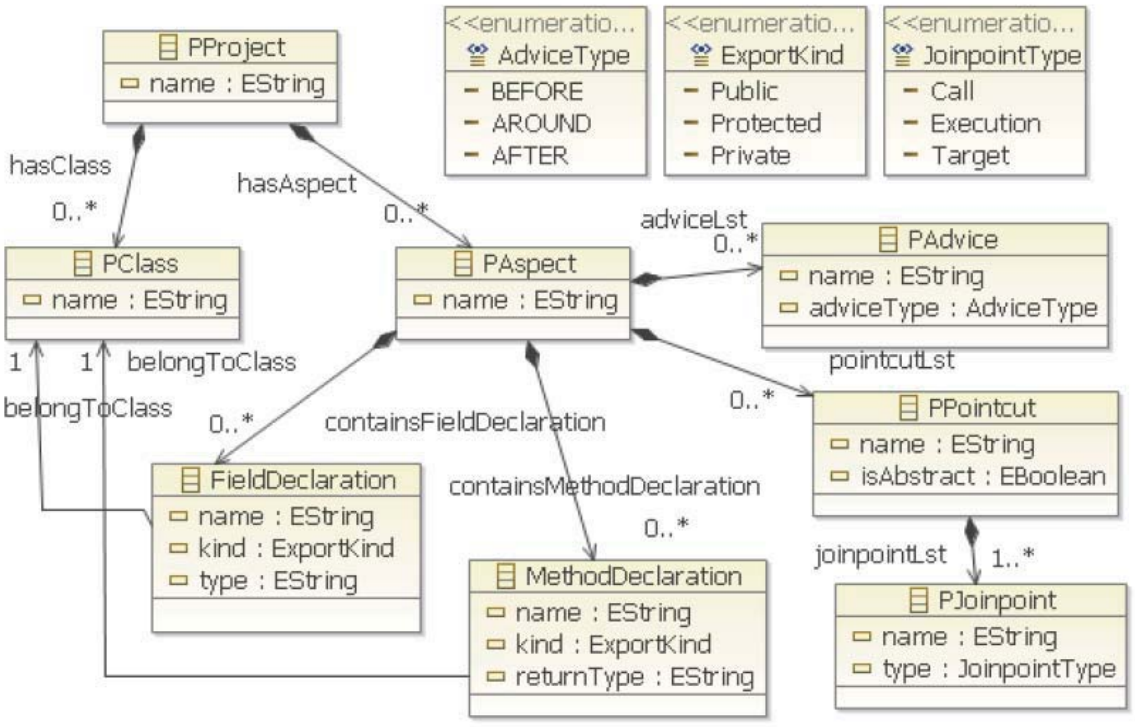

Figure 13: The proposed AspectInfo metamodel.

3.2 Proposing an AspectInfo Model

The AspectBench Compiler (abc) separates the aspect-specific information from the standard Java code in a given project, it generates in the frontend an AST from the plain Java code and an information construct from the AspectJ information. This aspect information construct is called *AspectInfo*, which we used as the base of the information we need from an AspectJ project. We propose a simple metamodel called `AspectInfo` metamodel for the information we get from the abc compiler. Figure 13 displays the `AspectInfo` metamodel.

3.2.1 The proposed AspectInfo metamodel in XMI Format

As we discuss in Section 3.3, we need to represent the proposed metamodel in a model interchange format in order to use it. We also discuss how we deploy the Eclipse Modeling Framework (EMF) to represent the metamodels as the `Ecore` files in the XML Metadata Interchange(XMI) [35] format. The XMI is an standard proposed by OMG for manipulating metadata information using Extensible Markup Language (XML).

To represent `AspectInfo` metamodel as an Ecore model, we use the `Ecore Diagram` Eclipse plug-in. With the help of this plug-in, we display the metamodel as a class diagram while the `Ecore` and `Genmodel` files are automatically generated from this class diagram.

Figure 14 represents the Ecore model for the `AspectInfo` metamodel.

3.3 Modeling Crosscutting Concerns for KDM

3.3.1 Extending KDM

In this section, we present and discuss our proposed AspectJ KDM profile that can support the development and maintenance of applications written in AspectJ.

There are two ways to extend the KDM metamodel:

1. Framework extension metamodel pattern.

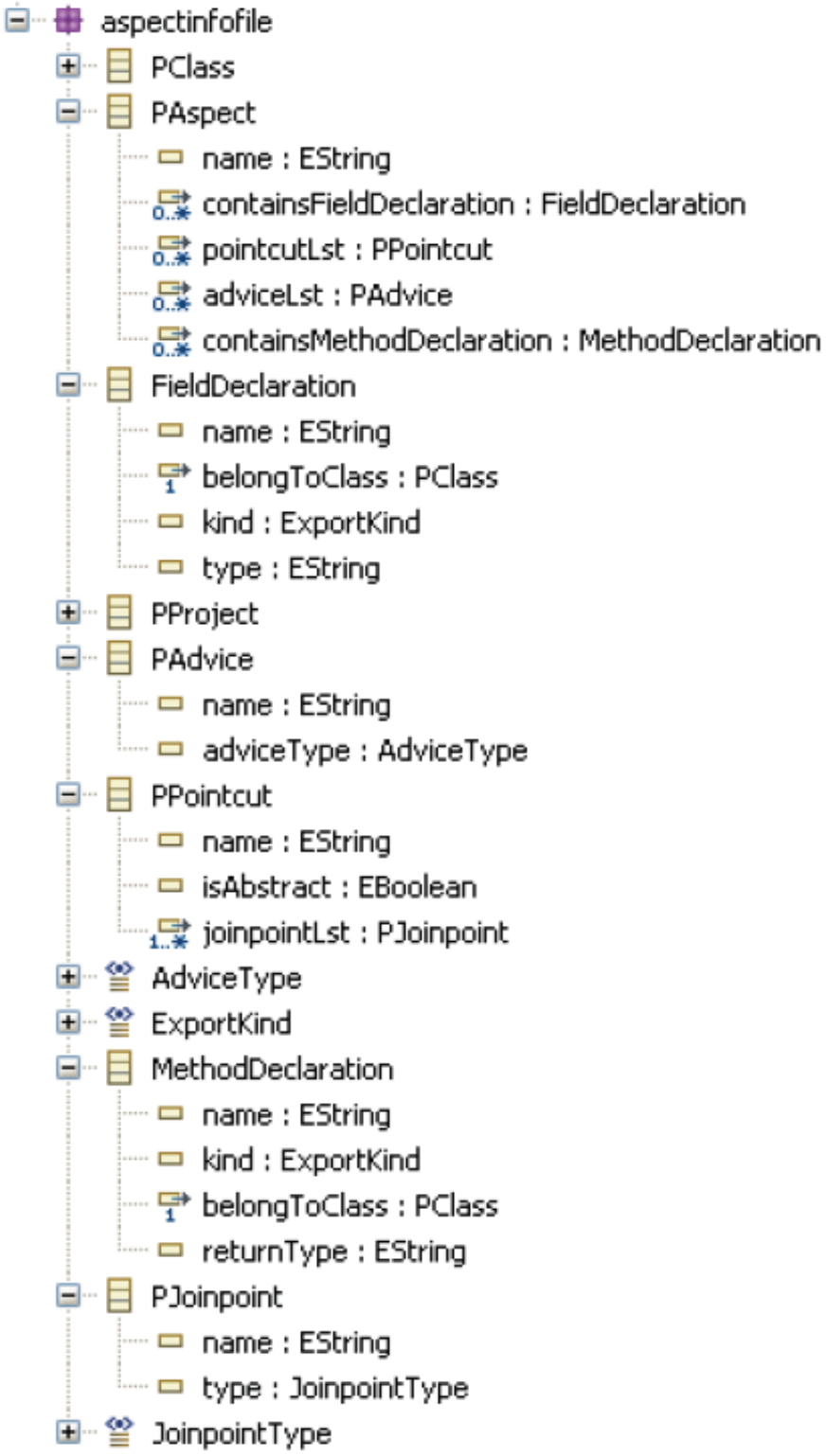

Figure 14: The AspectInfo metamodel in XMI format (Ecore).

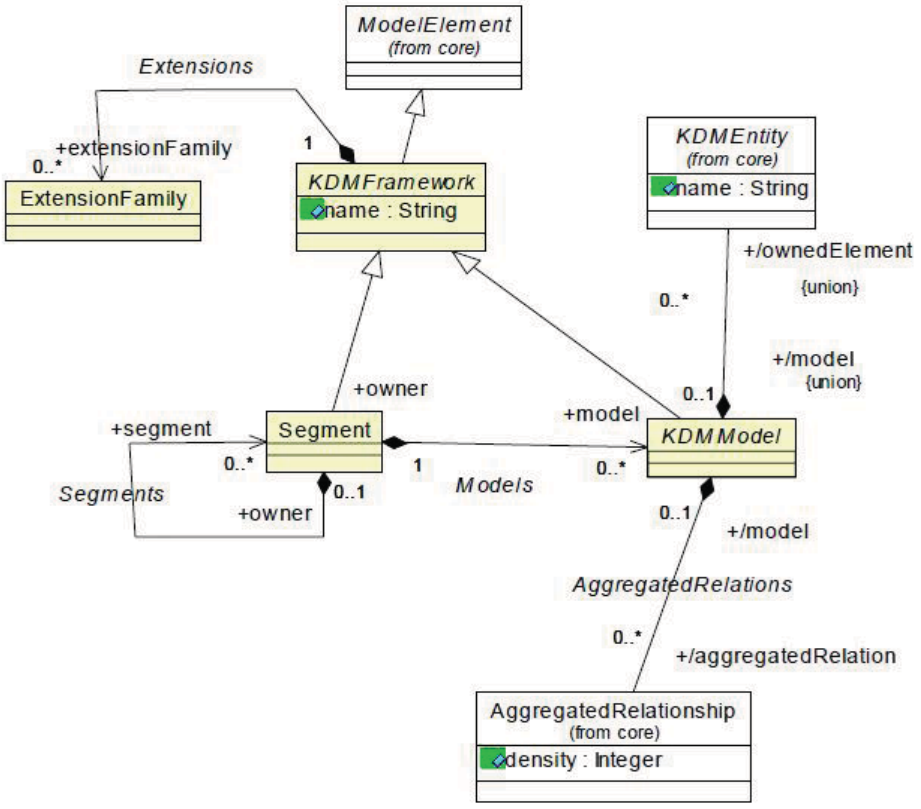

Figure 15: The KDM Framework class diagram. (Adopted from [36])

2. Lightweight extension mechanism.

We discuss these extension mechanisms in the following sections.

Framework Extension Metamodel Pattern

All KDM packages deploy this pattern to extend the `KDMFramework`. The `KDM Framework` class diagram is shown in Figure 15. The `KDMFramework` and the `KDMModel` are abstract classes. The `KDMModel` is the key unit of KDM instances. Each concrete

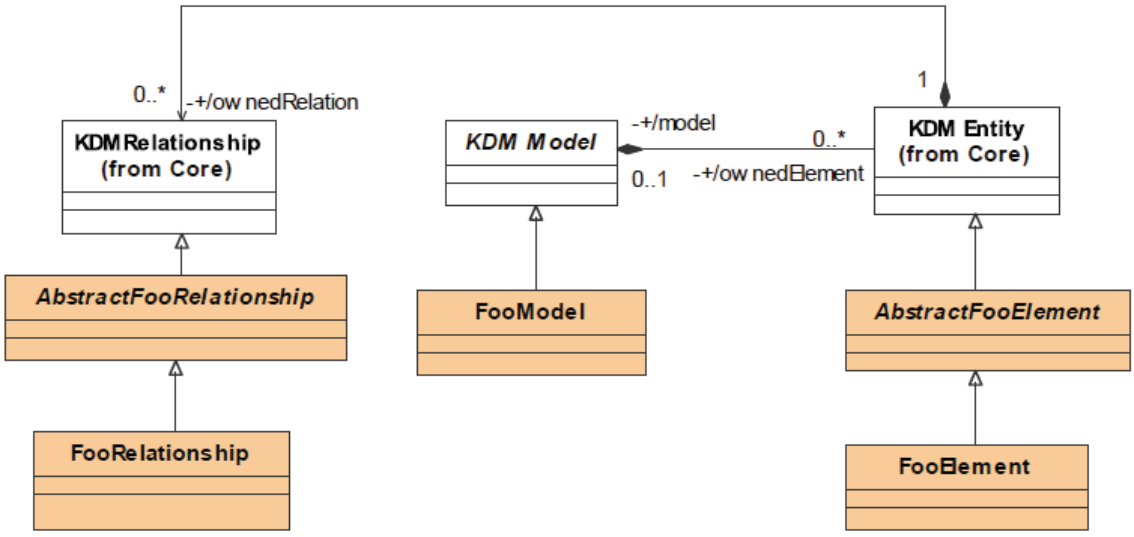

Figure 16: Foo model as an extension to KDM model. Shaded elements correspond to our proposed extensions.

KDM model instance deploy the framework extension metamodel pattern. By deploying this pattern, one can add a new KDM model in a new package to the KDM metamodel [36]. The framework extension mechanism has a naming convention; this means that the model inheriting from KDM model, KDM relationship and KDM element must change the word KDM with the name of the model they want to add to KDM metamodel. The following example (`Foo` example) from the KDM specification [36] specifies the framework extension pattern step by step and it also represents how naming convention works. Figure 16 illustrates the Foo model as an extension to KDM model and Figure 17 shows the Foo inheritance class diagram.

- We define a new package named `Foo` which contains the metamodel elements of Foo.

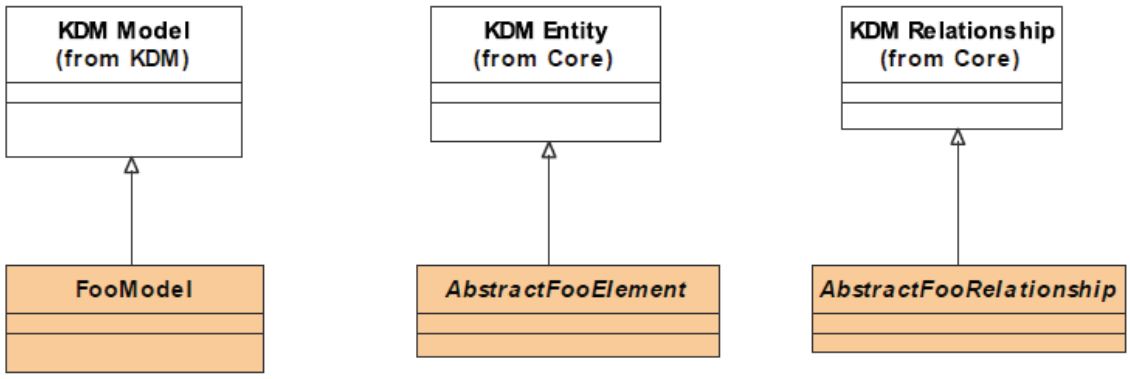

Figure 17: The Foo inheritance class diagram. Shaded elements correspond to our proposed extensions.

- In the `Foo` package, we define the `FooModel` as a subclass of `KDMModel`.
- In the `Foo` package, we define the `AbstractFooElement` as a subclass of `KDMEntity` class to be the abstract parent for all entities of the `Foo` model.
- In the `Foo` package, we define the `AbstractFooRelationship` as a subclass of `KDMRelationship` class to be the abstract parent for all relationship of the `Foo` model.
- `FooModel` owns `AbstractFooElement`. `AbstractFooElement` owns zero or more `AbstractFooRelationship`. These associations are subsets of the associations between their parent classes: `KDMModel`, `KDMEntity`, `KDMEntity` and `KDMRelationship`.

Lightweight Extensions Mechanism

This mechanism is defined in the `KDM Extensions` class diagram. The lightweight extension mechanism allows the introduction of stereotypes. The stereotype provides

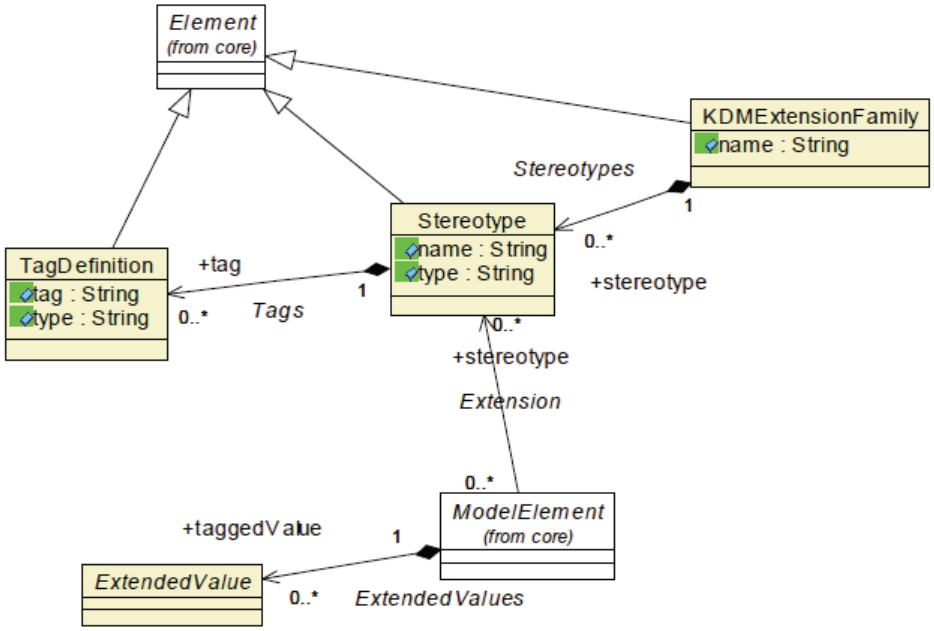

Figure 18: The KDM Extensions class diagram. (Adopted from [36])

additional semantics to the base metamodel element but it does not change the semantics of the base metamodel. Figure 18 shows the `KDM Extension` class diagram.

The following sequence of steps describes how to use the lightweight extension mechanism:

1. Define a set of stereotypes.

2. Define tag definitions which are the attributes of stereotypes.

3. Use the extended elements the same way as the base metamodel elements.

The Knowledge Discovery Metamodel supports the lightweight extension mechanism using Generic Element metamodel pattern. Generic metamodel elements are those which, through subclassification, can be used as *extension points*. In addition

to all the generic elements, each KDM model defines two generic elements that can be used as extension points: the generic entity and the generic relationship. Extension points with the most specific semantics (closer to the bottom of the class hierarchy) should be used as stereotypes.

The KDM program element layer contains a set of metamodel elements to represent the common constructs in programming languages (such as Java). As AspectJ is an extension to Java, its set of linguistic constructs is a superset of that of Java. Therefore, we use the lightweight extension mechanism to introduce stereotypes and extend the KDM metamodel with AspectJ elements.

3.3.2 The KDM AspectJ metamodel specification

The following steps describe how to deploy the KDM lightweight extension mechanism to support AspectJ concepts:

1. Adopt a domain model for AspectJ.

2. Mapping domain model elements to the KDM elements.

Adopting a Domain Model for AspectJ

The domain model contains a set of language constructs that capture the concepts of our domain. For aspect modeling, these concepts are aspect, advice, pointcut and join point. In this paper, we deploy the `CoreAOP` profile proposed by Sharafi *et al.* [46]

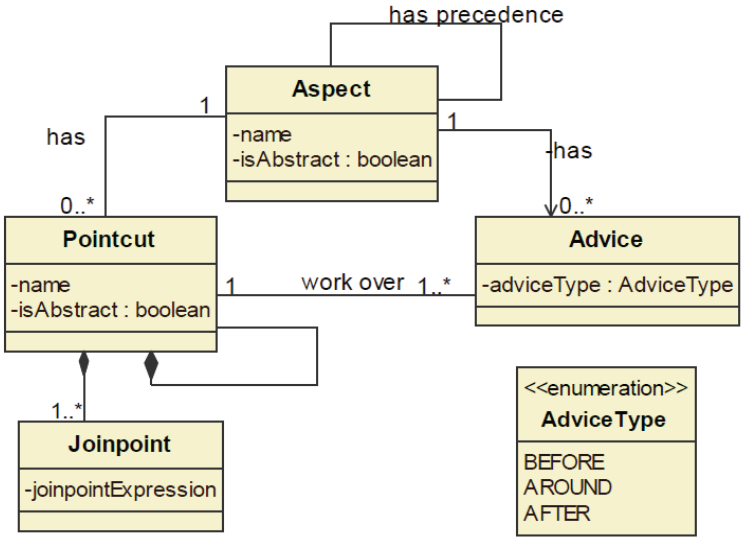

Figure 19: The Core AOP domain model. (Adopted from [46])

as a base metamodel for AspectJ representation (see Figure 19 for `CoreAOP` domain model). The most important characteristic of the proposed Unified Modeling Language (UML) profile [38] is the language independence property, which can be used to provide metamodels for other aspect-oriented languages. The detailed description of this profile can be found in [46]. Moreover, our proposed domain model for AspectJ is partially similar to Everman's [17]. Everman modeled join points by defining different kind of pointcuts. In contrast, we model join point construct separately from pointcuts.

In the following section, we describe how we map constructs presented in this domain model to KDM elements and extend KDM elements if necessary [36].

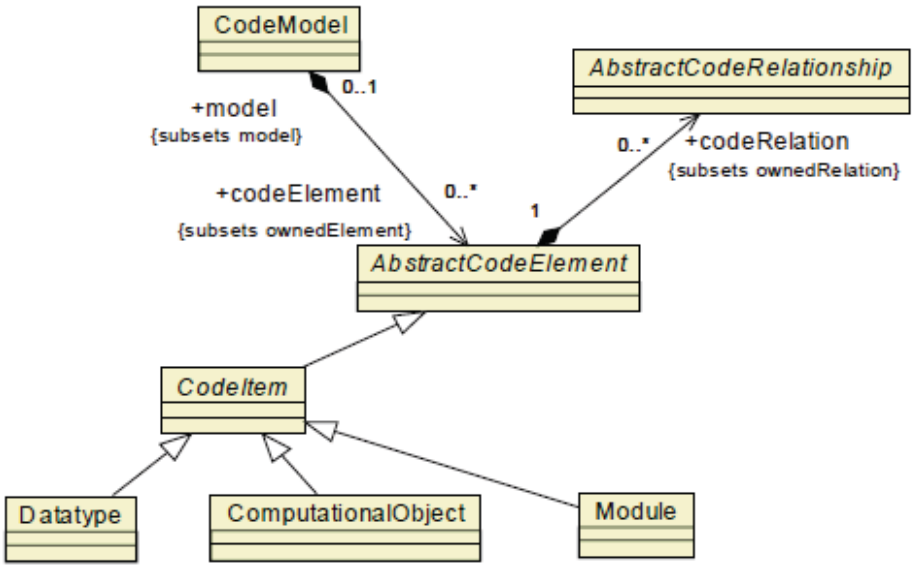

Figure 20: The Code Model class diagram. (Adopted from [36])

Mapping Domain Model Elements to KDM Elements

In this section, we discuss the mapping between AspectJ concepts and the KDM metamodel. Figure 20 shows the `CodeModel` class diagram from the KDM specification, which will be used during the mapping. Table 1 contains `CodeModel` class diagram elements and their specifications. We use elements from the `CodeModel` class diagram and extend them to add AspectJ constructs. The elements of the `AspectJ KDM` profile are presented in what follows while the descriptions of the stereotypes are listed in Table 2.

Aspect: An aspect is a unit of modularity that encapsulates static and dynamic features of a concern similar to the class definition. An aspect contains pointcuts (static feature) and advice blocks (dynamic feature). Additionally an aspect

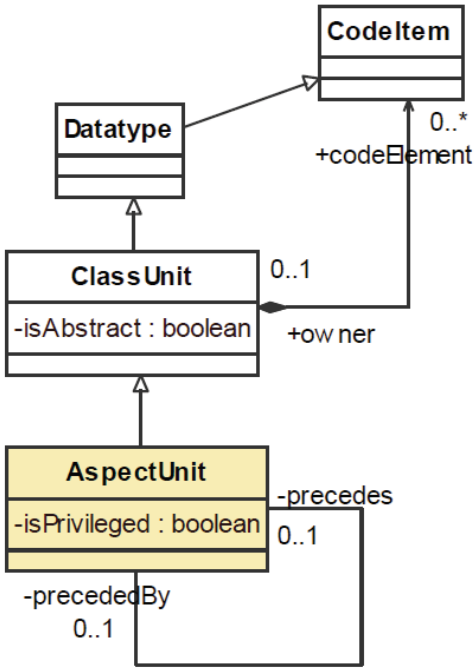

Figure 21: The AspectUnit in KDM metamodel. Shaded elements correspond to our proposed extensions.

definition can be inherited and much like a class, an abstract aspect can be deployed to enforce inheritance. These characteristics are close to the KDM metamodel `ClassUnit` so the meta-class `AspectUnit` extends the meta-class `ClassUnit`. Since the `AspectUnit` is a subclass of `ClassUnit`, it can have instances of the `ComputationalObject` like `MethodUnit` and `MemberUnit`. We believe that this way we can have the introduction in AspectJ as well. Figure 21 illustrates the `AspectUnit` in KDM metamodel.

Advice: An advice block encapsulates behavior. It indicates what happens whenever the program reaches specific points during its execution. There are three kinds of advice: before, after and around which can run before, after or in place of the

join point it operates over. We model advice as the `AdviceUnit` Class that is a `ControlElement` which represents the behavioral features of the `ClassUnit`, such as `MethodUnit` and `CallableUnit`. An advice is not semantically a method and it is not invoked explicitly. According to the KDM specification [36], the `MethodUnit` "represents member functions owned by a `ClassUnit`." Also the `MethodUnit` contains the `MethodKind` (constructor, destructor, virtual ...) as an attribute, which is not applicable to the advice. An advice is not a `CallableUnit` since `CallableUnit` has an enumeration called `CallableKind` (regular, external, and operator) as an attribute which is not applicable to the advice. Since the `ControlElements` are owned by the `ClassUnit`, the `AdviceUnit` does not need to be associated with the `AspectUnit`. The `ControlElement` is a superclass of this element. Figure 22 illustrates the `AdviceUnit` in KDM metamodel.

Pointcut: A pointcut is a predicate of join points. A pointcut can be given a name and it can be reused. Figure 23 illustrates the `PointcutUnit` and `Joinpoint` in KDM metamodel. We model pointcut as a `PointcutUnit` and we suggest that `PointcutUnit` extends the `MemberUnit` class because the `MemberUnit` class can be owned only by `ClassUnit`. By extending `MemberUnit` class the `PointcutUnit` class can be owned only by `ClassUnit` and `AspectUnit` as the result. Also the `PointcutUnit` has the same `ExportType` (visibility of the member such as public, private and protected) that the `MemberUnit` has. Since there is no association between subclasses of `ComputationalObject`, we have to add one between the `AdviceUnit` and the

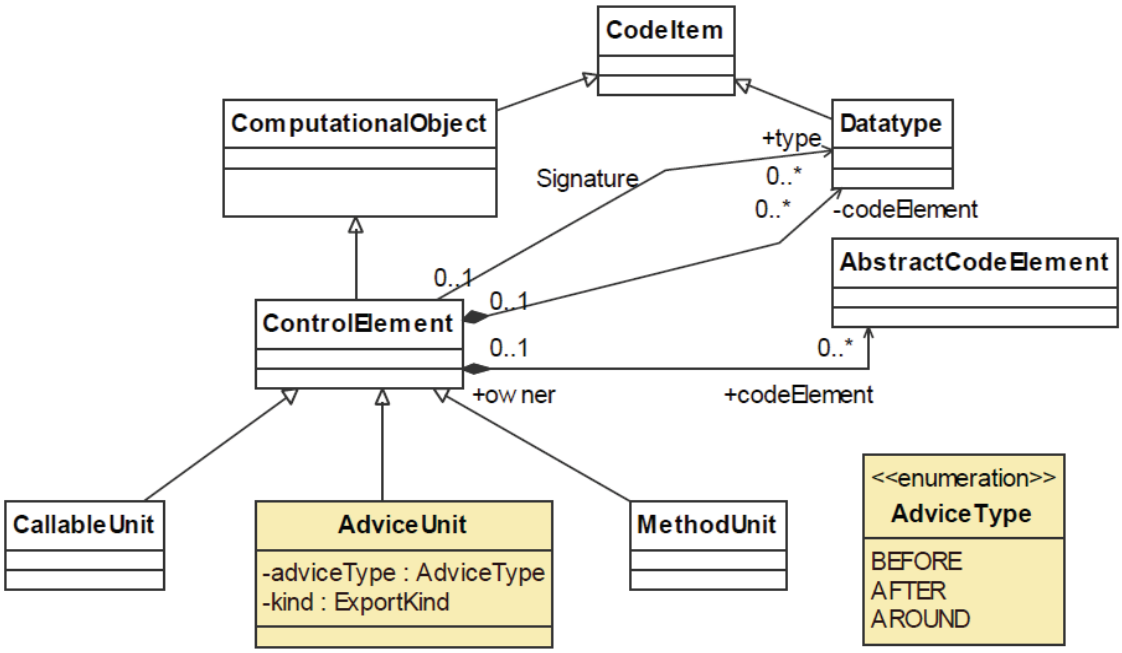

Figure 22: The AdviceUnit in KDM metamodel. Shaded elements correspond to our proposed extensions.

`PointcutUnit`. An advice must have a pointcut expression, and a pointcut may belong to more than one advice (this way pointcut definition makes reuse possible.) Only `ClassUnit`s that are stereotyped as `AspectUnit`s can have `MemberUnit`s stereotyped as `PointcutUnit`s and `ControlElement`s stereotyped as `AdviceUnit`.

Composite Pointcut: We use the **composite pattern** [20] to illustrate the relation between join point and pointcut. Figure 24 illustrates the composite pattern structure. A pointcut can be composed by possibly many join points. This way we may have any number of join points we want in a specific pointcut. The `PointcutUnit` conforms to the `Component`, the `JoinPoint` to the `Leaf`, the `CompositePointcut` to the `Composite` and the `AdviceUnit` to the `Client` in the composite pattern.

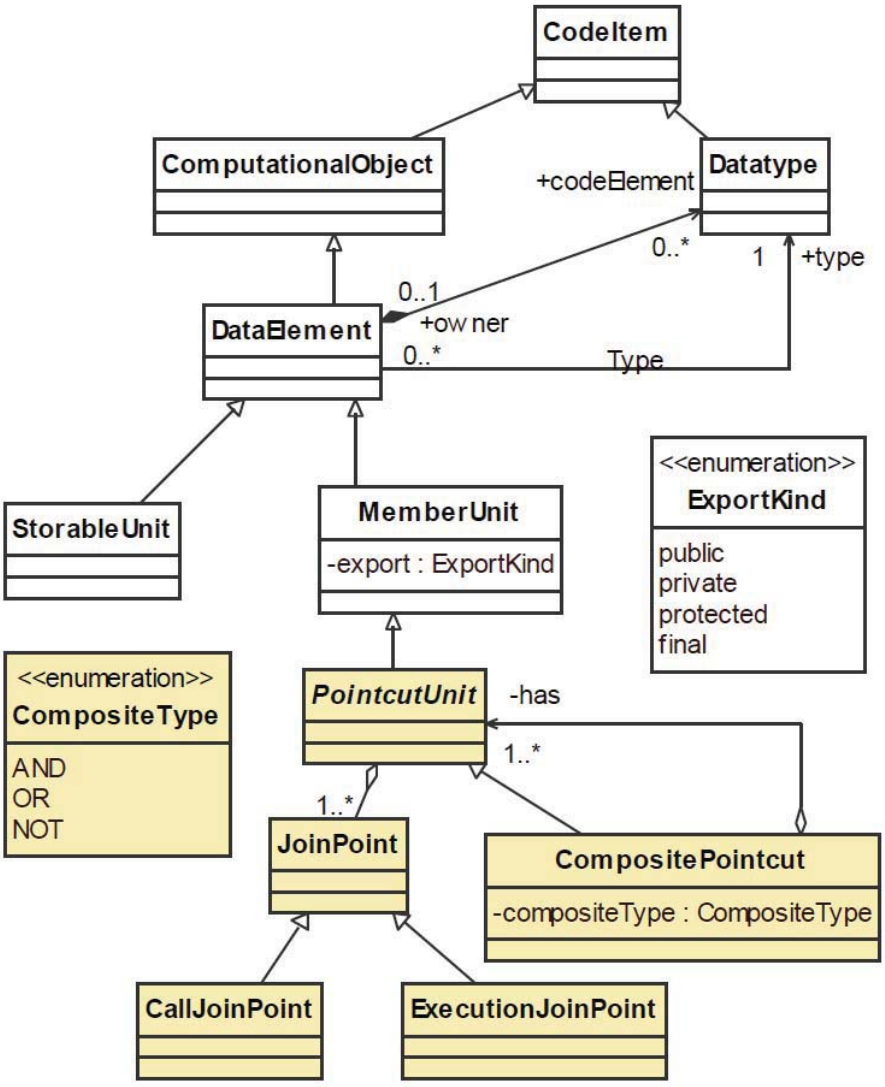

Figure 23: The PointcutUnit in KDM metamodel. Shaded elements correspond our proposed to extensions.

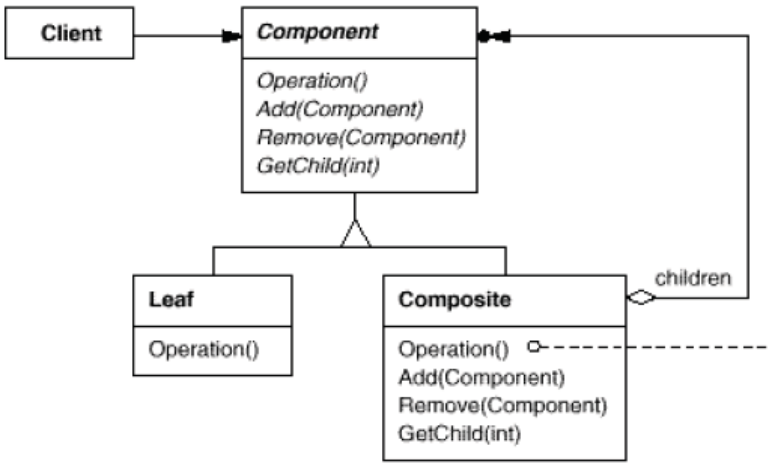

Figure 24: The composite pattern structure. (Adopted from [20])

`CompositePointcut` has an attribute `CompositeType`. The `PointcutUnit` is the superclass of this element.

Join point: A join point is a well-defined point in the execution of a program. The set of join points supported by AOP programming languages is referred to as the join point model. In AspectJ, method calls and method executions are common examples of join points. Supported join points can extend the `JoinPoint` Class. Figure 25 shows the KDM metamodel which includes the aspect-oriented constructs.

3.3.3 The proposed metamodel using a Model Interchange Format

To use the proposed metamodel we need a format to represent it. Sharing the metamodels between different tools and users requires a widely used, standard model

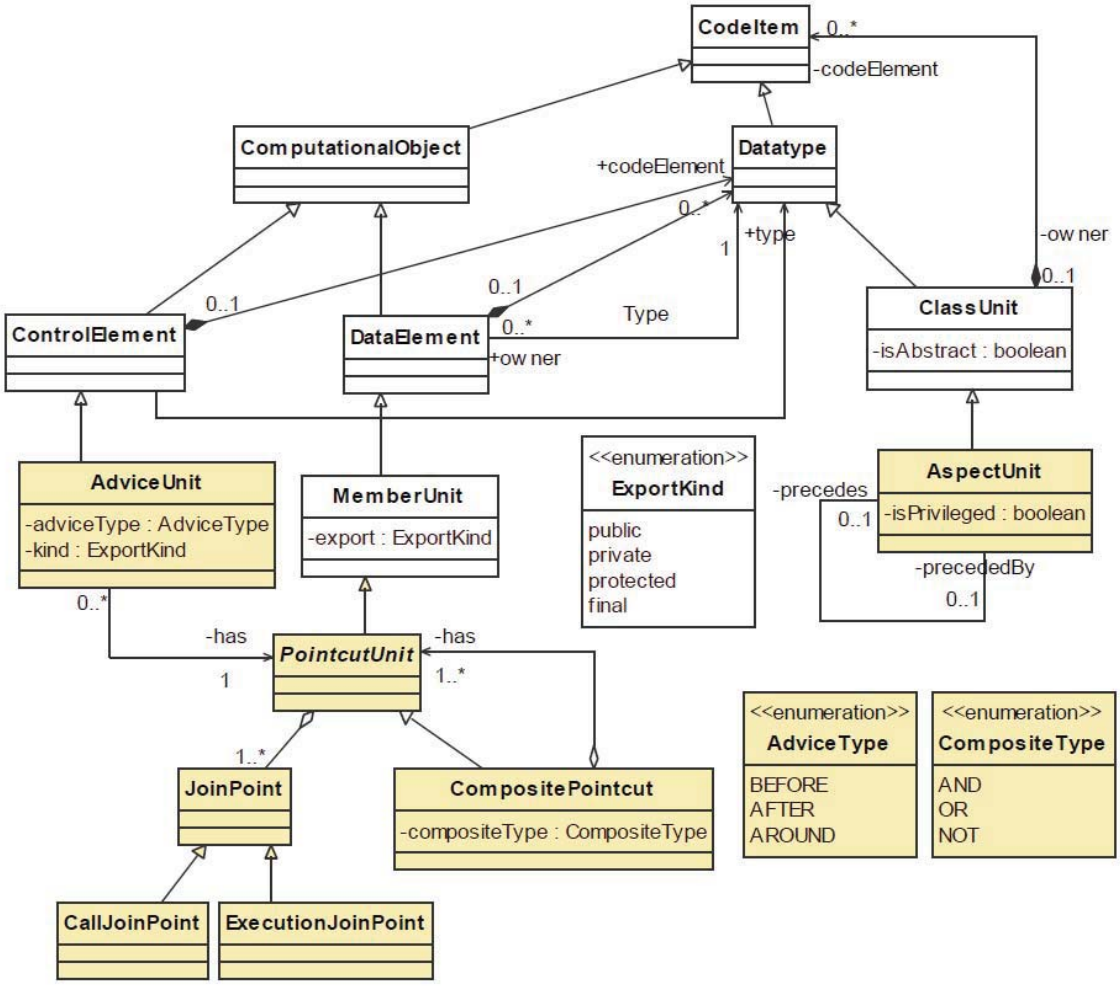

Figure 25: Adding aspect-oriented constructs to the KDM metamodel. Shaded elements correspond to our proposed extensions.

interchange format. St-Denis *et al.* [47] provide a list of model interchange formats (such as XMI [35], RDF [30] and RSF [52]) along with their advantages and disadvantages.

We adopt the XML Metadata Interchange (XMI) format in this project. The XMI format can be used to express any metamodel which is based on MOF. This format is based on XML. The XMI model interchange format fulfills most of the requirements for a good model interchange format such as transparency, simplicity and scalability [47]. The choice of using MOF and XML standards make XMI a flexible and evolvable solution. The XMI format also enjoys a wide industry support.

The Eclipse Modeling Framework (EMF) is a Java framework and code generation facility for standard models, representing the metamodels as `Ecore` files in XMI format. After specifying the AspectInfo and AspectKDM metamodels in EMF, we can define transformations and generate Java code for them.

There is a number of ways for specifying a metamodel in XMI format for EMF:

- Using an XML editor to write the metamodel in XMI format.
- Using Java annotation with model properties.
- Importing the XMI format made by a modeling tool such as IBM Rational Rose.
- Using XML schema (XSD) [18] to describe the metamodel.

The XML schema file for KDM metamodel is available at OMG KDM website [36]. The basic mapping from the `XML schema` constructs to the `Ecore` constructs are:

- A `schema` maps to an `EPackage` type.
- A `complex` type maps to an `EClass` type.
- A `simple` type maps to an `EData` type.
- An `attribute` maps to an `EAttribute` or an `EReference` depending on its type.

To generate the KDM Ecore from the KDM XML schema in Eclipse, we first should create a new EMF project and choose the XML schema as the model importer. An Ecore model (*KDM.ecore*) and a generator model (*KDM.genmodel*) is created after specifying the location of KDM XML schema file. To monitor the exact mapping from the XML schema to the Ecore, the *KDM.xsd2ecore* is created by selecting the "Create XML Schema to Ecore Map" in the process. The KDM Analytics and the MoDisco have generated the *KDM Ecore* model which can be found in their websites [3, 32].

To make the `Aspect KDM` metamodel from the KDM metamodel, we have to add the Aspectual elements to the *KDM Ecore*. The KDM metamodel elements that we use as base classes are all parts of the Program Element Layer in the KDM and belong to the `Code` package in the KDM Ecore. By choosing the `Code` package, we can find the "add new child" options which can be:

- EAnotation
- EClass
- EData Type
- EEnum
- EPackage

As illustrated in Figure 25, the new metamodel elements are all EClasses or EEnums:

- EClass: AspectUnit, AdviceUnit, PointcutUnit, CompositePointcut, Joinpoint, CallJoinpoint, ExecutionJoinpoint.
- EEnum: AdviceType, CompositePointcutType.

For each EClass, we can specify the ESuperType attribute. This attribute determines whether the class is abstract or it is an interface in the property view or not. For example, the AspectUnit is neither an abstract class nor an interface so we set these attributes to false, and the ClassUnit is its ESuperType.

To add the relationships and attributes between the Ecore elements, we choose a specific class and then "add a child" options. The following options are available:

- EAnnotation
- EOperation

- `EAttribute`
- `EReference`

After adding an `EReference`, we can specify its name, multiplicity, whether it represents a composite relationship or not, and the `EClass` at the other end of the relationship. The attribute type can be specified for an `EAttribute`. For example, we add `hasPointcut` to the `CompositePointcut`, which is a composite relationship and `PoincutUnit` is the `EClass` at the other end of the relationship. The `CompositePointcut` has an attribute called `compositeType` which is of the type `CompositePoincutType EEnum`.

Figure 26 illustrates the Ecore model for the `AspectKDM` metamodel.

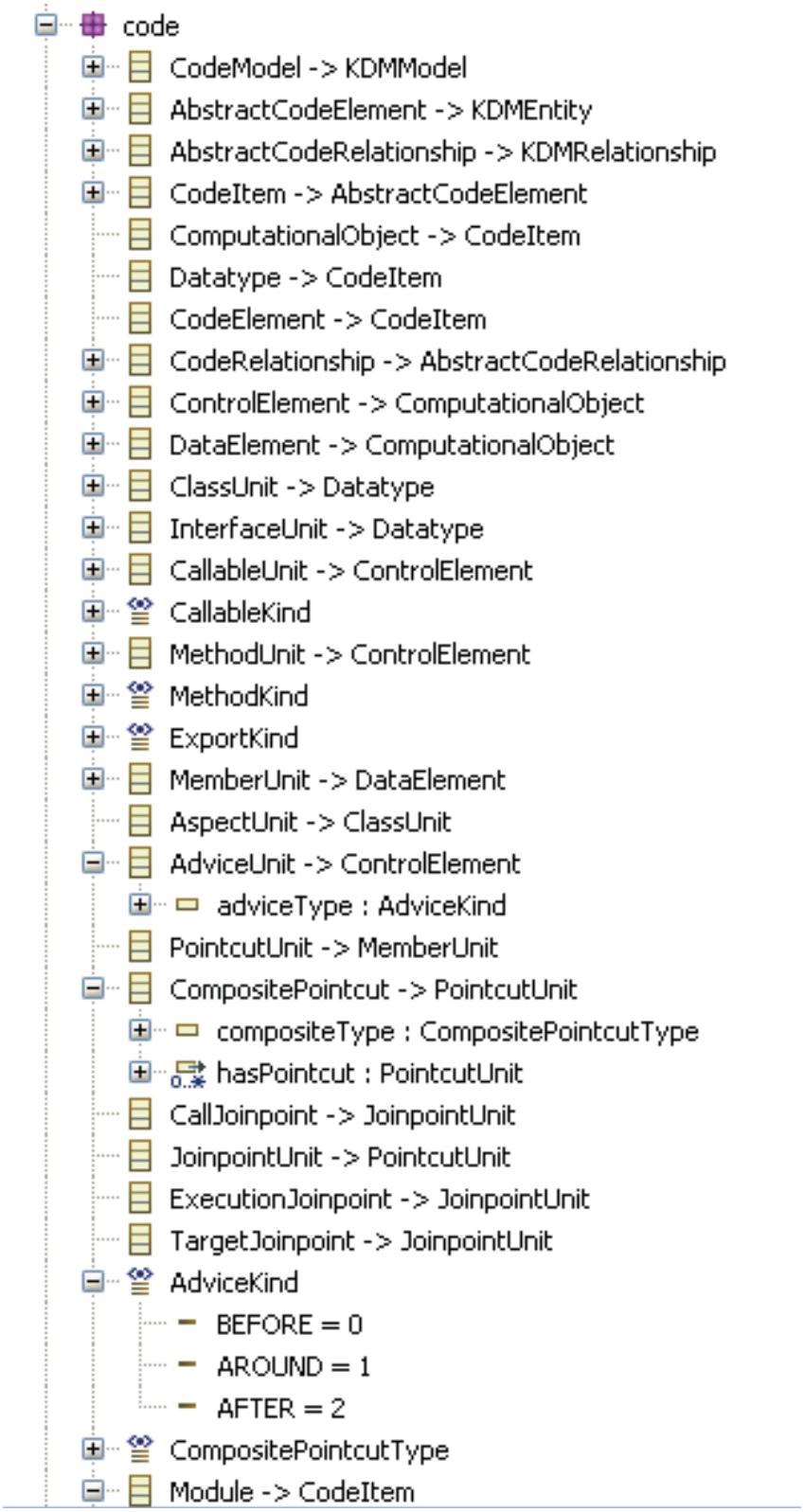

Figure 26: The Aspect KDM metamodel in XMI format (Ecore).

```
module fileAspect2kdmAspect;
create OUT : KDMAspect from IN : FileAspect;
```

Figure 27: ATL transformation module.

3.4 Model transformations

In this Section, we describe the transformations used to map the `AspectInfo` model to the `KDMAspect` model. The name after keyword `module` is the transformation name and it should correspond to the file name of the transformation. The module name here is `fileAspect2kdmAspect`. The target and source models are introduced by `create` and `from` keywords. The declaration of the target and source models must conform to the form `model name` : `metamodel name`. For example in our case (see Figure 27) the target model name is `OUT` and the target metamodel name is `KDMAspect`, and the source model name is `IN` and the source metamodel name is `FileAspect`.

A `matched rule` is introduced by the keyword `rule`. Each `matched rule` has two mandatory parts called source and target patterns which come after the keywords `from` and `to`. The source pattern specifies which elements from the source model are matched and the target pattern specifies which element(s) should be generated after a source element is matched.

The `PAspect2AspectUnit` rule (see Figure 28) is the most important rule of the transformation, since aspect is the most important element that owns other

```
rule PAspect2AspectUnit {
      from sfAspect : FileAspect!PAspect
      to kdmAspectUnit : KDMAspect!AspectUnit (
            name <- sfAspect.name,
            codeElement <- sfAspect.pointcutLst,
            codeElement <- sfAspect.adviceLst,
            codeElement <- sfAspect.containsMethodDeclaration,
            codeElement <- sfAspect.containsFieldDeclaration )
}
```

Figure 28: Aspect transformation rule.

```
rule PAdvice2AdviceUnit {
      from  sfAdvice : FileAspect!PAdvice
      to    kdmAdviceUnit : KDMAspect!AdviceUnit (
            name <- sfAdvice.name,
            adviceType <- sfAdvice.adviceType   )
}
```

Figure 29: Advice transformation rule.

information about crosscutting concerns like the pointcuts and advice. This rule aims to generate `AspectUnit` of the `KDMAspect` metamodel from the `PAspect` of the `FileAspect`. The name of the `AspectUnit` is the same as the name of the `PAspect`. After that, the corresponding list of elements in `KDMAspect` should be generated from the lists of pointcuts, advice, and method and field declarations. In the target metamodel, the `KDMAspect`, since elements like `PointcutUnit` and `AdviceUnit` are subclasses of the `CodeItem` metaclass then their relationship with `AspectUnit` is through `codeElement`, the relationship of the metaclasses `CodeItem` and `Class`. In `PAspect2AspectUnit`, it is enough to specify the name of the relationship that the `AspectUnit` metamodel has with `PointcutUnit` and `AdviceUnit`, and etc. ATL itself matches and calls the appropriate rules for matching the specified lists.

```
rule MethodDeclaration2MethodUnit {
      from  sfMethodDeclaration : FileAspect!MethodDeclaration
      to    kdmMethodUnit : KDMAspect!MethodUnit (
            name <- sfMethodDeclaration.name,
            export <- sfMethodDeclaration.kind,
            codeElement <- sfMethodDeclaration.belongToClass)
}
```

Figure 30: Method declaration transformation rule.

```
rule FieldDeclaration2MemberUnit {
      from  sfFieldDeclaration : FileAspect!FieldDeclaration
      to    kdmMemberUnit : KDMAspect!MemberUnit (
            name <- sfFieldDeclaration.name,
            export <- sfFieldDeclaration.kind,
            codeElement <- sfFieldDeclaration.belongToClass )
}
```

Figure 31: Field declaration transformation rule.

The `PAdvice2AdviceUnit` (see Figure 29) transforms `PAdvice` into `AdviceUnit` by mapping the name and the advice type.

The method and the field declarations are examples of Introduction in AspectJ. We have decided to model the introduced method and field by an aspect, like a regular one in the KDM metamodel, only with the difference that their owner is an aspect. In the `MethodDeclaration2MethodUnit` (see Figure 30) the `declared method` in the `FileAspect` is mapped into the `MethodUnit` in the `KDMAspect`. In the `FieldDeclaration2MemberUnit` (see Figure 31) the declared field in `FileAspect` is mapped into the `MemberUnit` in `KDMAspect`. The method and field are declared as the members of a class, this class is declared at the last line of each transformation.

In the `PPointcut2CompositePointcut` (see Figure 32) each `pointcut` is mapped into a `CompositePointcut`.

```
rule PPointcut2CompositePointcut {
      from  sfPointcut : FileAspect!PPointcut
      to    kdmCompositePointcut : KDMAspect!CompositePointcut (
      name <- sfPointcut.name,
            hasPointcut <- sfPointcut.joinpointLst    )
}
```

Figure 32: Pointcut transformation rule.

```
rule PJoinpoint2CallJoinpoint {
      from  sfJoinpoint : FileAspect!PJoinpoint
                              (sfJoinpoint.type = #Call)
      to    kdmCallJoinpoint : KDMAspect!CallJoinpoint (
                  name <- sfJoinpoint.name      )
}
```

Figure 33: Call join point transformation rule.

```
rule PJoinpoint2ExecutionJoinpoint {
      from  sfJoinpoint : FileAspect!PJoinpoint
                               (sfJoinpoint.type = #Execution)
      to    kdmExecutionJoinpoint : KDMAspect!ExecutionJoinpoint (
                  name <- sfJoinpoint.name      )
}
```

Figure 34: Execution join point transformation rule.

```
rule PClass2ClassUnit {
      from  sourceFileClass : FileAspect!PClass
      to    kdmClasstUnit : KDMAspect!ClassUnit (
                  name <- sourceFileClass.name  )
}
```

Figure 35: Class transformation rule.

In the source metamodel, each `join point` has a `type` which indicates the type of join point; for example the `call`, the `execution` and etc. On the other hand, in the target metamodel `JoinpointUnit` is a class which has different type of join points as its subclasses (for example `CallJoinpoint`, `ExecutionJoinpoint`, etc). To map one class in the source metamodel to its multiple corresponding classes in the target metamodel, we use the guard in the source pattern part of the transformation. In the `PJoinpoint2CallJoinpoint` (see Figure 33) and `PJoinpoint2ExecutionJoinpoint` (see Figure 34) the guard in the source pattern specifies the `join point type` in the source metamodel. If it is of type `call` then it will be mapped to a `CallJoinpoint` and if it is of type `execution`, it will be transformed into the `ExecutionJoinpoint`.

The last rule is the `PClass2ClassUnit` which maps the `class` in the source metamodel into the `ClassUnit` in the target metamodel.

3.5 Summary

The objective of this Chapter was to create KDM representation from an AspectJ project. In this chapter we have proposed:

1. AspectKDM metamodel.

2. AspectInfo metamodel.

3. A set of model-to-model transformation from AspectInfo to AspectKDM.

The AspectKDM metamodel is a KDM metamodel for AspectJ family of AOP languages. The AspectInfo metamodel represents the information we need from the AspectJ project. If we change the compiler the output of the compiler should be an instance of this metamodel. For using the AspectKDM and AspectInfo metamodels we need to capture them in a specific, formal and persistent format. We have deployed XMI format for this purpose. Finally, we define a set of model-to-model transformations from AspectInfo to AspectKDM.

Table 1: The definition of the elements of the CodeModel class diagrams.

Class name	Specification
CodeModel	It is a specific KDM model and the only model of the Program Elements Layer. It owns some information about the software system such as code.
AbstractCode Element	It is an abstract class representing any generic elements a programming language contains.
AbstractCode Relationship	It is an abstract class representing any relationship a programming language contains.
CodeItem	It represents the named elements of a programming language.
Computational Object	It represents a computational object at runtime (such as procedures and variables).
Datatype	It represents datatypes of a programming language.
ClassUnit	It represents the user-defined classes in object-oriented languages. Attributes: • isAbstract:Boolean - The indicator of an abstract class. Associations: • codeElement:CodeItem [0..*] - The list of class members.
ControlElement	It is a common superclass for callable code elements such as methods and functions.
DataElement	It is a common superclass for storable data items such as global and local variables.
MemberUnit	It represents a member of a class type. Attributes: • export: ExportKind - Represents the visibility of the member such as public, private and protected. Constraint: • MemberUnit can be owned only by a ClassUnit.

Table 2: KDM AspectJ metamodel stereotype specification.

Class name	**Base metaclass**	**Specification**
AspectUnit	ClassUnit	Attributes: • isPrivileged:Boolean Associations: • precedes:AspectUnit[0..1] Indicates whether an aspect has an immediate precedence over another aspect. • precededBy:AspectUnit[0..1] Indicates whether another aspect has an immediate precedence over this aspect.
AdviceUnit	Control Element	Attributes: • isPrivileged:Boolean Associations: • has: PointcutUnit [1] - Association to the pointcut. Constraint: • The name of AdviceUnit should be the same as the name of the PointcutUnit it is attached to. • The AdviceUnit can only be owned by the ClassUnits stereotyped as AspectUnit.
PointcutUnit	MemberUnit	Attributes: • isAbstract:Boolean Associations: • belongTo: AdviceUnit [0..*] A pointcut may belong to zero or more advice. Constraint: • A pointcut must have a name.
Composite Pointcut	PointcutUnit	
JoinPoint	PointcutUnit	

Chapter 4

Case Study

Prior to discussing a case study, we disscuss how to set up an AspectJ project to discover its KDM representation.

4.1 The XMI Representation of an AspectJ project

The activity diagram in Figure 36 describes how the proposed metamodel for `AspectKDM` and `AspectInfo` can be used to model an AspectJ project using XMI format. We describe in Chapter 3.2 and Chapter 3.3 how to generate the `Ecore` files from the `AspectKDM` and `AspectInfo` metamodel. A `genmodel` file is created automatically from the `Ecore` file and it can be used to provide the three Eclipse plug-ins: the `Model code`, the `Edit`, and the `Editor`. By generating the `Model Code` plug-in, the interfaces for classes are created. It also creates two packages called `impl` and `util`

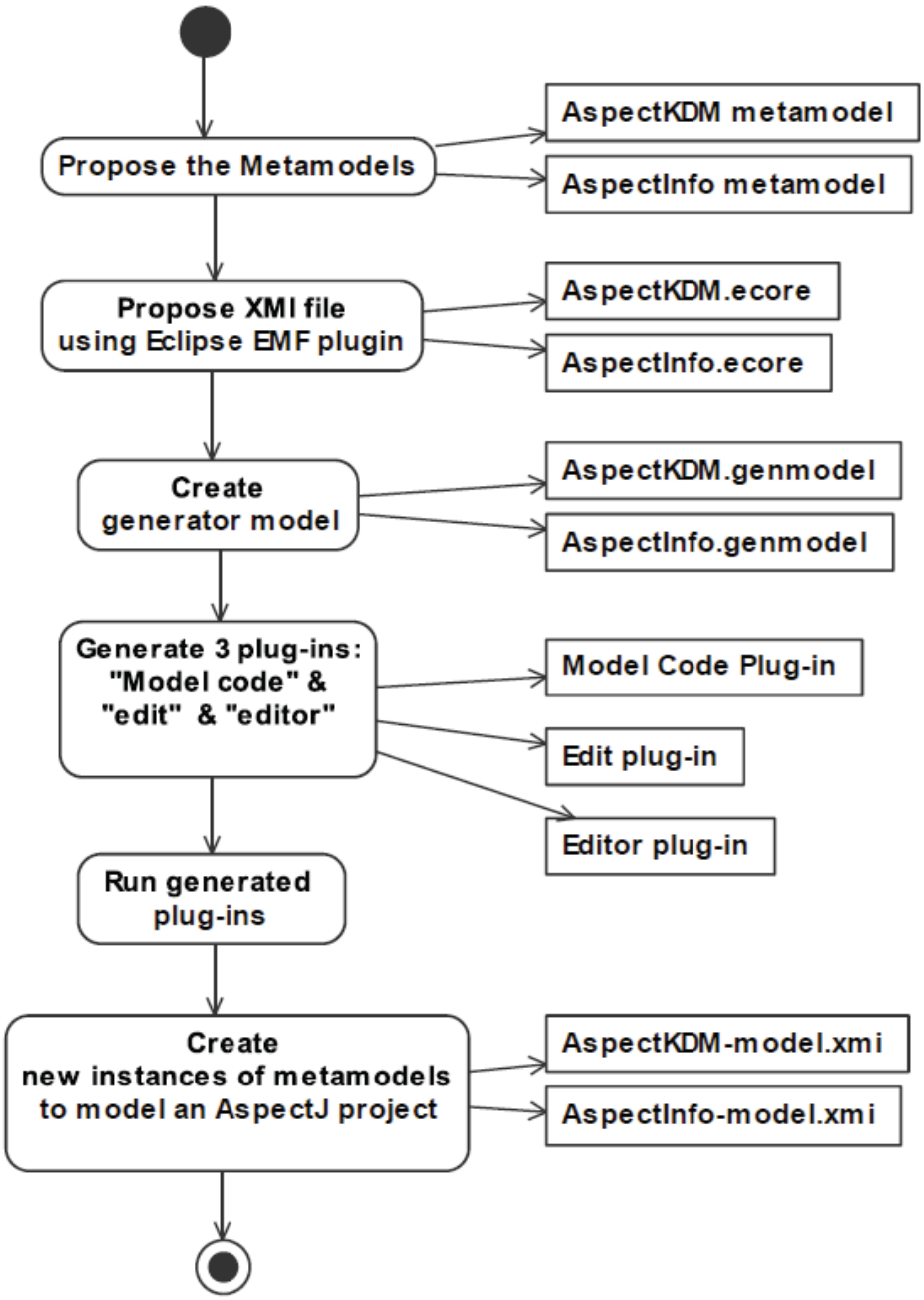

Figure 36: Using the proposed metamodels to model an AspectJ project in XMI format in EMF.

which include the implementation for the interfaces and some utility classes. The `Edit` plug-in provides a flexible layer between the `Model code` and the EMF `Editor`. It provides a structured view by using adaptors and it also provides support for command-based editing of a model. The `Editor` plug-in provides additional model-specific UI for the editor and wizards. Together the `Edit` and `Editor` plug-ins can provide a fully functional Eclipse editor for any model.

By running the generated plug-ins together, the proposed metamodels are added to the models that are provided by EMF. By opening a new EMF project, it is possible to create a new instance model of `AspectKDM` and `AspectInfo` metamodels by using the `model wizard` in Eclipse. The newly created instance model is opened in the main view in Eclipse and it is possible to add the model elements from the `AspectKDM` or `AspectInfo` to it.

4.2 Case Study: Telecom

To demonstrate the applicability of our proposal, we choose a telecommunications simulation as a case study, `Telecom`, is provided as part of the Eclipse AspectJ Development Tools package [49]. The program simulates local and long distance connections between two customers in a telecommunications system. Telecom is an AspectJ Benchmark in the literature with three interesting aspects; it also deploys all the elements defined in the metamodel. Figure 37 illustrates the UML class diagram

for this system. This system contains the following classes:

- **Customer:** The customer is the caller or receiver of a call. It is known by its `name` and the `area code`. The customer manages the call with the protocols such as `call`, `pickup`, and `hang up`.
- **Connection:** The connection is the circuit between two customers which can be local or long distance.
- **Call:** The process in which a customer tries to connect to another customer is a call. A call is held between two customers by creating a connection between them.
- **Timer:** The timer simulates a simple timer for calculating the time of each connection.
- **Abstract Simulation:** It is responsible for executing the system, connecting customers, and providing a complete report of all customers' activities. The `Abstract Simulation` has three subclasses:
    1. **Basic Simulation:** It simulates the execution of a program by implementing the `AbstractSimulation.run(..)` method.
    2. **Billing Simulation:** It provides the bill which contains connection time for each customer by implementing the `AbstractSimulation.report(..)` method.

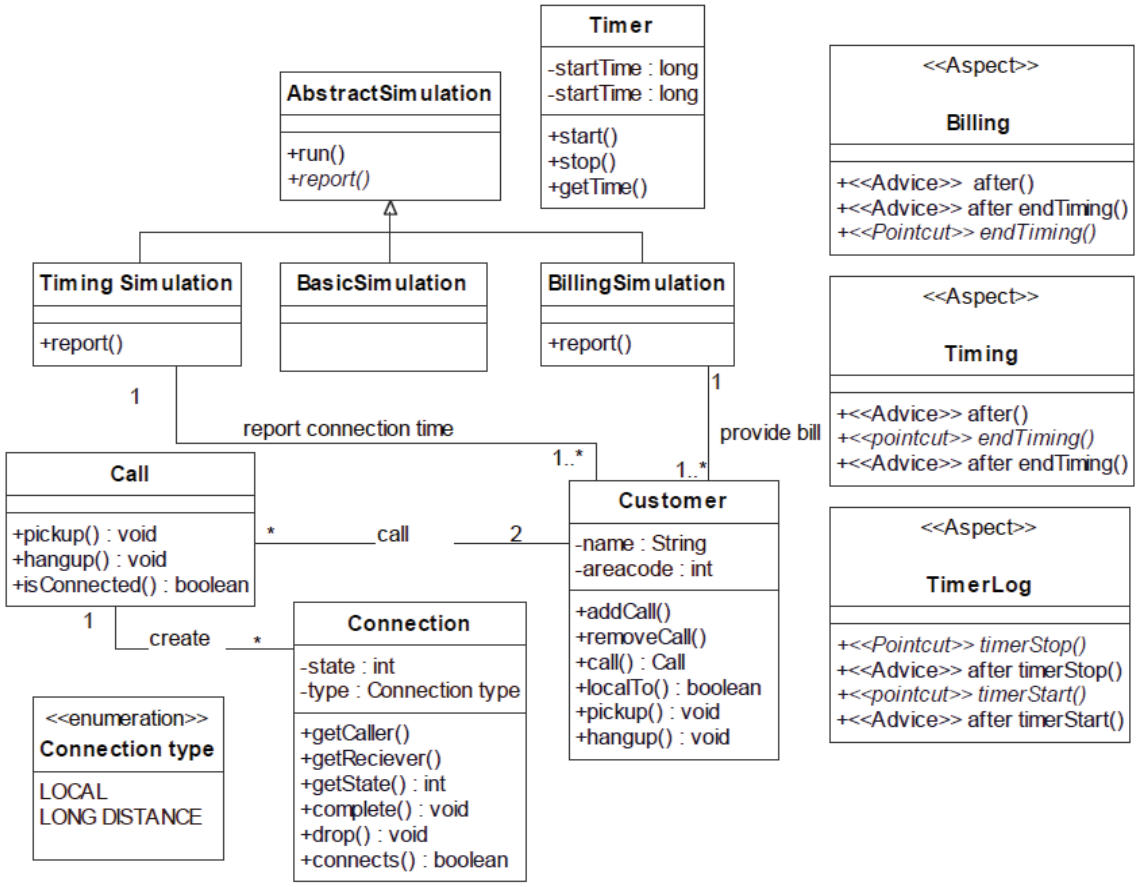

Figure 37: The telecom system class diagram. (Adopted from [46])

3. **Timing Simulation:** It prints a report of the connection time by implementing the `AbstractSimulation.report(..)` method.

This system contains three aspects to provide billing service for customers and logging the activities:

1. The `Timing` aspect calculates the time of each call for each customer.

2. The `TimerLog` aspect logs the Timer class activities.

3. The `Billing` aspect provides a complete bill for each customer including the expenses based on type of connection and time of the calls.

We work through the case study as follows: we deploy the abc compiler to extract AspectJ elements from the three `.aj` files. Then, using EMF, we instantiate the `AspectInfo` metamodel to provide models of these three aspects.

Figures 38, 39, 40 illustrate respectively the `Timing`, `Billing`, and `TimerLog` aspects as `AspectInfo` models.

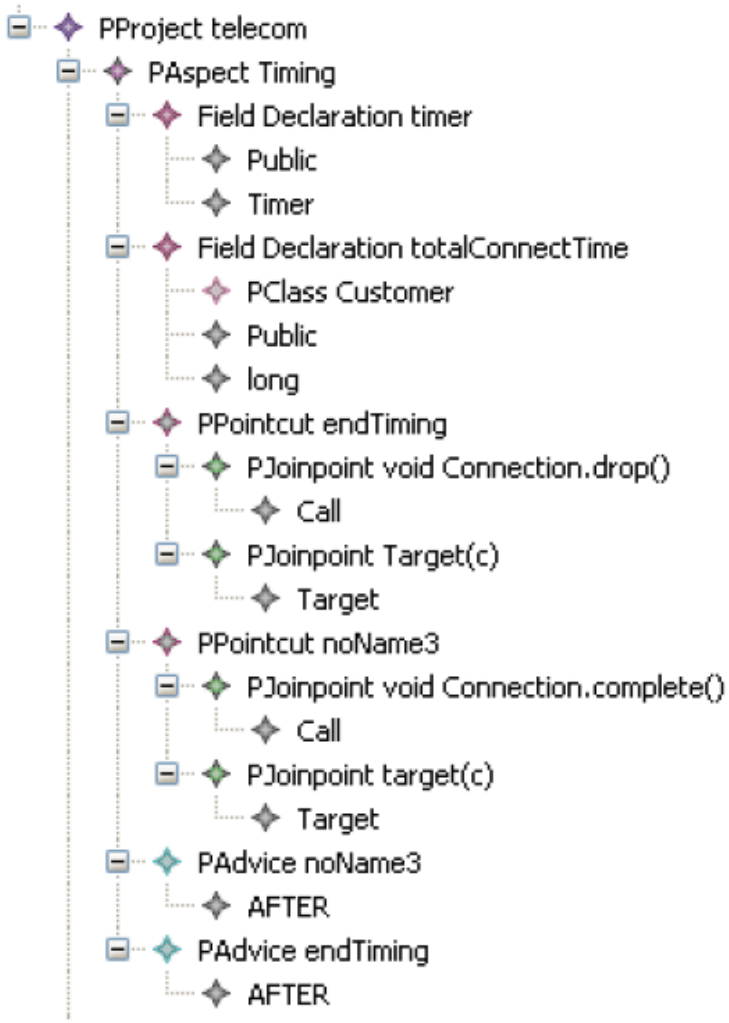

Figure 38: The Timing aspect modeled as an AspectInfo model.

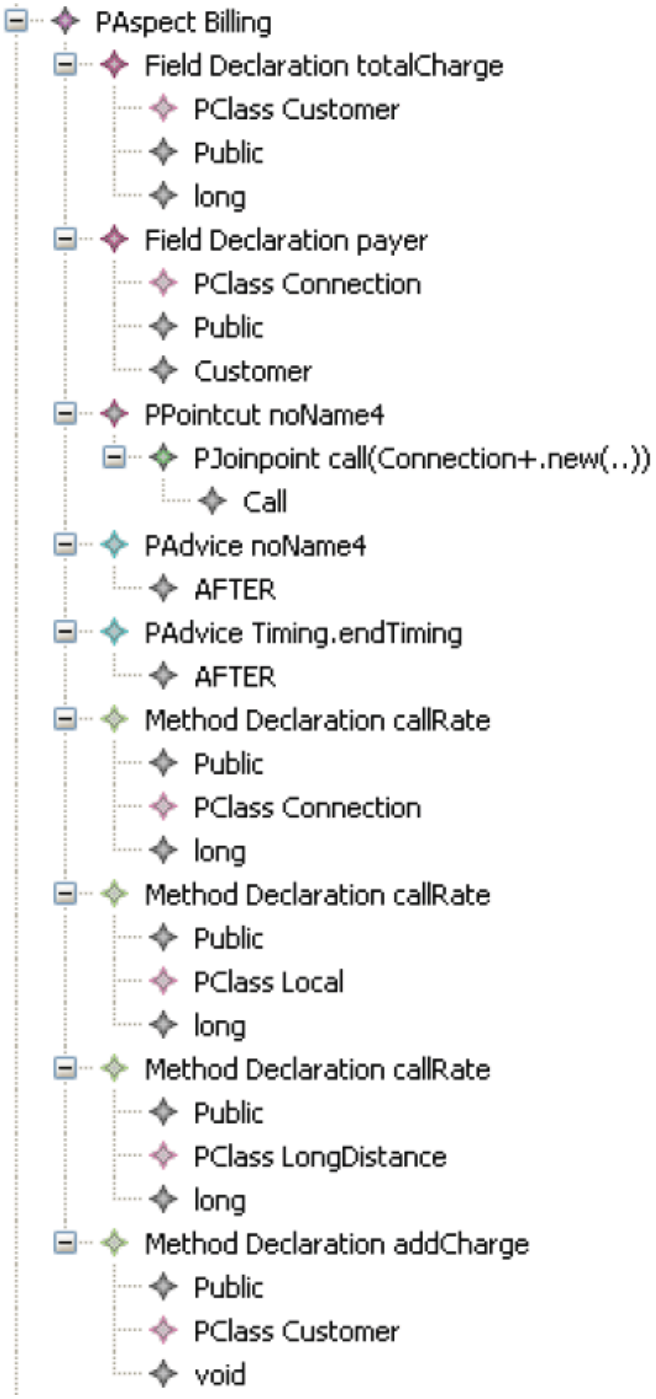

Figure 39: The Billing aspect modeled as an AspectInfo model.

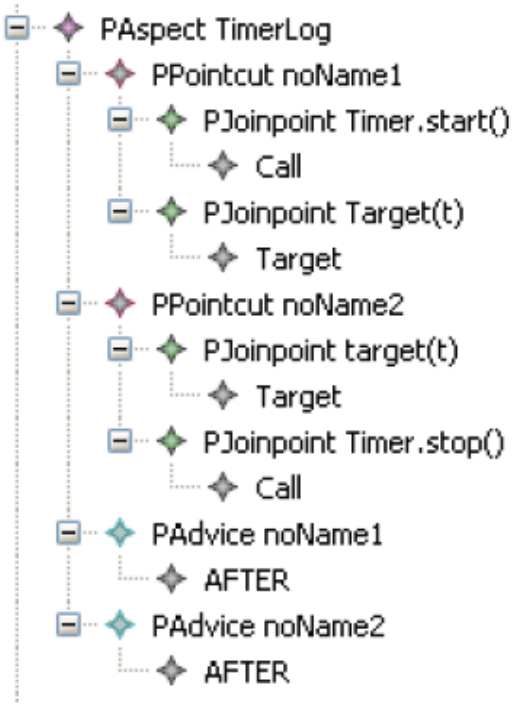

Figure 40: The TimerLog aspect modeled as an AspectInfo model.

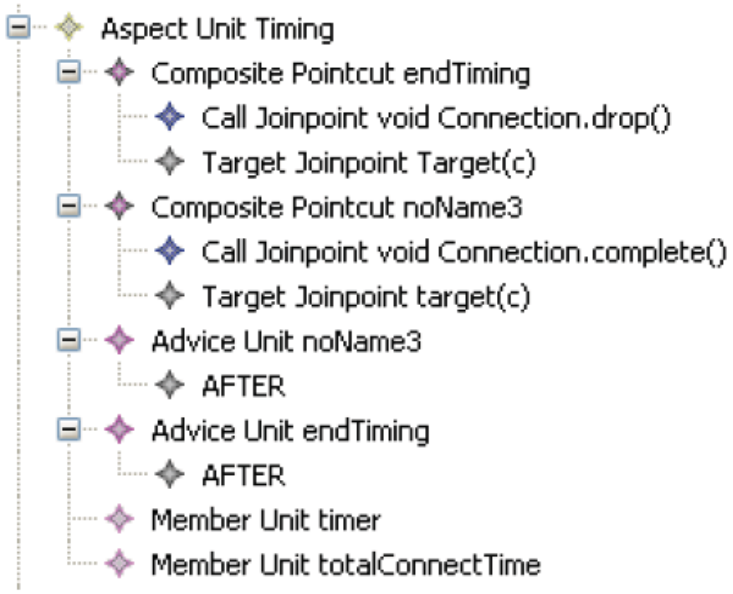

Figure 41: The Timing aspect modeled as an AspectKDM model.

Finally we apply ATL transformation to map `AspectInfo` model to `AspectKDM` model. Figures 41, 42, 43 represent the `AspectKDM` model for `Timing`, `Billing`, and `TimerLog` aspects respectively.

The Telecom example is an AspectJ Benchmark in the literature. It has deployed every elements defined in the metamodel. The KDM representation of this case study has the elements from the AspectKDM metamodel with the correct use of syntax and semantics. It can be used by the KDM-compatible tools for different purposes such as analysis and maintenance.

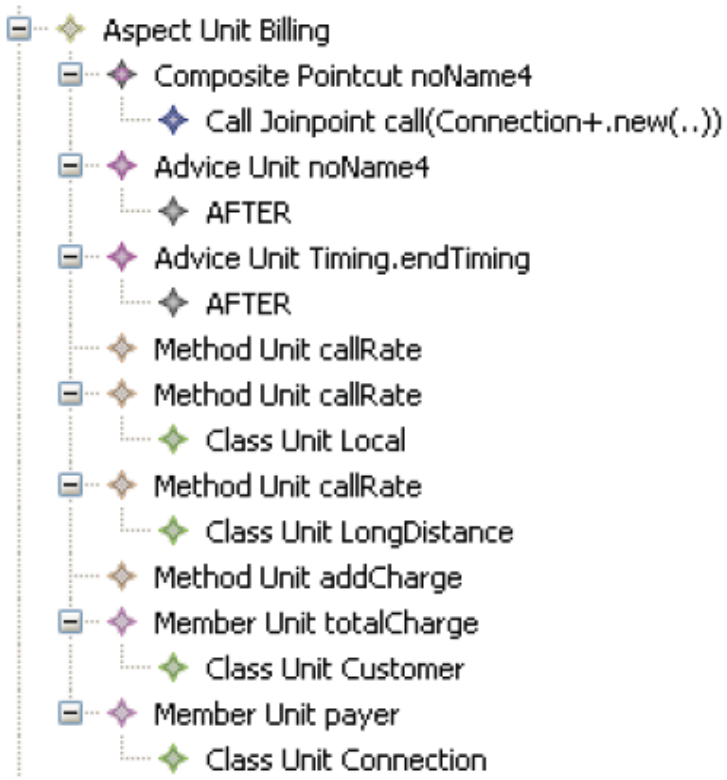

Figure 42: The Billing aspect modeled as an AspectKDM model.

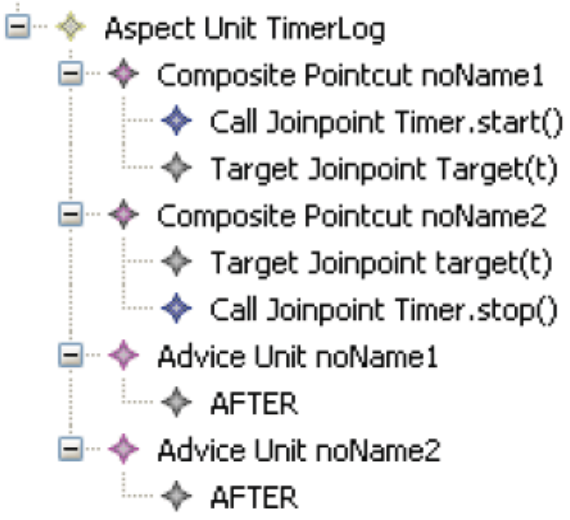

Figure 43: The TimerLog aspect modeled as an AspectKDM model.

Chapter 5

Conclusion

The Knowledge Discovery Metamodel (KDM) provides a common representation of programs written in a number of programming languages. This representation can utilize the deployment of maintenance tools. In the literature, research work has defined a support for crosscutting concerns in UML either through heavyweight extensions (extending the UML profile) or lightweight extensions (defining a profile for crosscutting concerns). No support has been proposed in the literature for crosscutting concerns in KDM.

In this thesis we proposed an extension (a metamodel) to the KDM to support the integration of the AspectJ programming language into the model specification. Moreover, we proposed a AspectInfo metamodel for the information we need from the AspectJ code and set of ATL model transformations to map AspectJ constructs to KDM elements.

It is important to note that the two releases of AspectJ (abc and AJDT) are not fully compatible (see for example the study in [4]). Because of this, a possible limitation of our work is that it is confined to the language grammar of abc.

We have provided automation and tool support through a collection of Eclipse plug-ins and Java projects, and we demonstrated the applicability of our approach through a case study.

As is, our proposal supports those AspectJ constructs that correspond to concepts that are necessary and sufficient to describe a system as aspect-oriented, namely join points, pointcuts and advice, all three of which are provided by most aspect systems. The choice of AspectJ has been based on the rationale that it is a general-purpose language with high popularity, tool support and an increasing community of researchers and practitioners.

Our proposal defines a proof of concept and our tool support is rather prototypical. For an industrial-strength automation and tool support, possible future developments should include the provision of support for other AspectJ constructs such as parent declaration and aspect precedence rules, as well as the improvement of automation and tool support through an integrated environment.

The KDM representation for AspectJ projects can be used the same as any other KDM representation for analysis purposes. As we mentioned earlier the ultimate benefit of KDM is to provide interoperability among different tools for various maintenance and evolution tasks. It enhances the maintenance of a software system due

to the existence of a uniform representation of programs that can be read by all KDM-compatible analysis tools.

Bibliography

[1] slides from a talk on abc: An extensible aspectj compiler. http://abc.comlab.ox.ac.uk/documents/architecture.pdf, 2004.

[2] abc. The AspectBench Compiler. http://aspectbench.org.

[3] KDM Analytics. http://kdmanalytics.com/kdm/index.php.

[4] V. Arnaoudova, L.M. Eshkevari, E.S. Sharifabadi, and C. Constantinides. Overcoming comprehension barriers in the AspectJ programming language. *Journal of Object Technology*, 7:121–142.

[5] Pavel Avgustinov, Aske Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc : An extensible aspectj compiler. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development*

I, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer Berlin / Heidelberg, 2006.

[6] G. Barbier, H. Brunelière, F. Jouault, Y. Lennon, and F. Madiot. MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*, page 365, 2010.

[7] E. Barra, G. Génova, and J. Llorens. An approach to Aspect Modelling with UML 2.0. In *Proceedings of the 5th International Workshop on Aspect-Oriented Modeling (AOM) at AOSD'04 Conference*, 2004.

[8] M. Basch and A. Sanchez. Incorporating aspects into the UML. In *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling (AOM) at AOSD'03 Conference*, 2003.

[9] Jean Bezivin, Mikael Barbero, and Frederic Jouault. On the applicability scope of model driven engineering. In *Proceedings of the International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 3–7. IEEE Computer Society, 2007.

[10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco:

a generic and extensible framework for model driven reverse engineering. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[11] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[12] C. Constantinides and T. Skotiniotis. Providing multidimensional decomposition in object-oriented analysis and design. In *Proceedings of the IASTED International Conference on Software Engineering, Innsbruck, Austria*, pages 17–19, 2004.

[13] V. Dehlen, F. Madiot, and H. Bruneliere. Representing Legacy System Interoperability by Extending KDM. In *Proceedings oF Model-Driven Modernization OF Software Systems*, page 96, 2008.

[14] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[15] Architecture Driven Modernization (ADM). http://adm.omg.org/.

[16] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44:29–32, October 2001.

[17] Joerg Evermann. A meta-level specification and profile for aspectj in uml. In *Proceedings of the 10th International Workshop on Aspect-oriented Modeling (AOM) at AOSD'07 Conference*, pages 21–27, New York, NY, USA, 2007. ACM.

[18] D.C. Fallside and P. Walmsley. XML schema part 0: primer second edition. *W3C recommendation*, 2004.

[19] L. Fuentes and P. Sánchez. Elaborating UML 2.0 profiles for AO design. In *Proceedings of 8th International Workshop on Aspect-Oriented Modeling (AOM) at 5th AOSD Conference, Bonn, Germany*, 2006.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[21] A. Gerber, E. Glynn, A. MacDonald, MJ. Lawley, and K. Raymond. Modeling for knowledge discovery. In *Proceedings of the Workshop on Model-Driven Evolution of Legacy Systems (MELS) at the EDOC'04 Conference, Monterey, USA*. IEEE Computer Society Digital Library, 2004.

[22] John Grundy and Rakesh Patel. Developing software components with the uml, enterprise java beans and aspects. In *Proceedings of Australian Software Engineering Conference (ASWEC'01)*, pages 127–136, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[23] Maria-Eugenia Iacob, Maarten W. A. Steen, and Lex Heerink. Reusable model transformation patterns. *IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'08), Los Alamitos, CA, USA*, pages 1–10, 2008.

[24] J. Irwin, G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, and J. Loingtier. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object Oriented Programing (ECOOP '07), IEEE, Finland*, pages 220–242, 1997.

[25] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[26] F. Jouault and I. Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg, 2006.

[27] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1188–1195, New York, NY, USA, 2006. ACM.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. In Jorgen Knudsen, editor, *Proceedings of the 15th European Conference on Object Oriented Programing (ECOOP*

'01), volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001.

[29] J. Kienzle, Y. Yu, and J. Xiong. On composition and reuse of aspects. In *Proceedings of the 2nd Workshop on Foundation of Aspect-Oriented Languages (FOAL '03) at AOSD '02 Conference, Boston, MA*, pages 17–24, 2003.

[30] Ora Lassila, Ralph R. Swick, World Wide, and Web Consortium. Resource Description Framework (RDF) Model and Syntax Specification, 1998.

[31] S.J. Mellor, S. Kendall, A. Uhl, and D. Weise. *MDA distilled*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

[32] MoDisco. http://www.eclipse.org/gmt/modisco/.

[33] B. Moyer. Software Archeology. Modernizing Old Systems. *Embedded Technology Journal*, 2009.

[34] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[35] OMG. XML Metadata Interchange (XMI) 2.1.1 Specification. Technical Report formal/2007-12-01, http://www.omg.org/spec/XMI/2.1.1/, 2007.

[36] OMG. Architecture-Driven Modernization (ADM) / Knowledge Discovery Meta-model (KDM) 1.2 Specification. Technical Report formal/2010-06-03, http://www.omg.org/spec/KDM/1.2/, 2010.

[37] OMG. Object Constraint Language (OCL) 2.2 Specification. Technical Report formal/2010-02-01, http://www.omg.org/spec/OCL/2.2/, 2010.

[38] OMG. Unified Modeling Language (UML) 2.3 Specification. Technical Report formal/2010-05-03, http://www.omg.org/spec/UML/2.3/, 2010.

[39] OMG. Meta Object Facility (MOF) 2.0-Query/View/Transformation (QVT) 1.1 Specification. Technical Report formal/2011-01-01, http://www.omg.org/spec/QVT/1.1/, 2011.

[40] OMG. Meta Object Facility (MOF), v2.4 - beta 2 Specification. Technical Report dtc/2010-12-08, http://www.omg.org/spec/MOF/2.4/Beta2/, 2011.

[41] Object Management Group (OMG). http://www.omg.org/.

[42] R. Pérez-Castillo, I.G.R. de Guzmán, M. Piattini, B. Weber, and Á.S. Places. An Empirical Comparison of Static and Dynamic Business Process Mining. In *Proceedings of the 26th Symposium On Applied Computing (SAC'11)*, TaiChung, Taiwan, 2011.

[43] R. Pérez-Castillo, I. García-Rodríguez de Guzmán, O. Ávila-García, and M. Piattini. MARBLE: A Modernization Approach for Recovering Business Processes

from Legacy Systems. In *Proceedings of the International Workshop on Reverse Engineering Models from Software Artifacts (REM'09)*, pages 17–20, 2009.

[44] R. Pérez-Castillo, I. García-Rodríguez de Guzmán, M. Piattini, and Á.S. Places. A case study on business process recovery using an e-government system. *Software: Practice and Experience*, 2010.

[45] A.M. Reina, J. Torres, and M. Toro. Towards developing generic solutions with aspects. In *Proceedings of the 5th International Workshop on Aspect-Oriented Modeling (AOM) at the UML 2004 Conference*, Lisbon, Portugal, 2004.

[46] Z. Sharafi, P. Mirshams, A. Hamou-Lhadj, and C. Constantinides. Extending the UML Metamodel to Provide Support for Crosscutting Concerns. In *Proceedings of the 34th ACIS International Conference on Software Engineering Research, Management and Applications (SERA'10), Montreal, Canada*, pages 149–157. IEEE, 2010.

[47] Guy St-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a model interchange format: The spool case study. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, Los Alamitos, CA, USA*, 2000.

[48] D. Stein, S. Hanenberg, and R. Unland. Designing aspect-oriented crosscutting in UML. In *Proceedings of International Workshop on Aspect-Oriented Modeling (AOM) with UML at AOSD'02 Conference, Enschede, Netherlands*, 2002.

[49] AspectJ Development Tools. http://www.eclipse.org/ajdt/.

[50] W. Ulrich. A status on OMG architecture-driven modernization task force. In *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems (MELS), Monterey, USA*. IEEE Computer Society Digital Library, 2004.

[51] Raja Valle-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the 9th international conference on Compiler construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer Berlin / Heidelberg, 2000.

[52] K. Wong. *Rigi users manual, version 5.4. 4*. The Rigi Group, University of Victoria, Victoria, Canada, 1998.