A Platform-independent Aspect-oriented Model and Patterns to Support Model Transformations

Zohreh Sharafi Tafreshi Moghaddam

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
August 2010

© Zohreh Sharafi Tafreshi Moghaddam, 2010

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-71052-4
Our file Notre référence
ISBN: 978-0-494-71052-4

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

A Platform-independent Aspect-oriented Model and Patterns to Support Model Transformations

Zohreh Sharafi Tafreshi Moghaddam

Model Driven Architecture (MDA) separates application logic from specific implementation technology to improve the reusability, portability and maintainability of the software system. However, current software system also needs to deal with other important concerns that are called crosscutting concerns that explicitly addressed by Aspect-oriented Programming (AOP). In this dissertation, we propose a model-driven approach to assess the benefits of AOP for MDA in order to provide increased modularity and to support related quality attributes. Even though research has been conducted toward modeling crosscutting concerns, these approaches found to be either language dependent or provide no support for aspectual behavior. This work has two contributions. First, we complement current works by proposing a language-independent extension to the UML metamodel to explicitly capture crosscutting concerns. The second contribution is to provide well-defined and automated model transformations to work with different models at various levels of abstraction and preserve their consistency.

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 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, Parisa Mirshams 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. I also wish to thank my dear friend, Pouya Jabbari for proof reading this dissertation.

Lastly, and most importantly, I owe my loving and sincere thanks to the most influential people in my life: my parents, Mahnaz Jafaripour and Aliakbar Sharafi for unconditional support and encouragement to pursue my interest. My twin sister, Azadeh, for being always beside me, listening to me and believing in me.

Contents

List of Figures

Chapter 1

Introduction

In 1974, Edsger W. Dijkstra in his paper "On the role of scientific thought" [10] talked about the *Separation of Concerns* (SoC) principle in computer society. At this time, SoC is one of the key principles in software engineering. The principle states that each system involves different kinds of concerns that should be identified and treated separately in order to deal with the complexity of a system, and in order to obtain the required engineering quality factors such as robustness, maintainability, and reusability.

Aspect-oriented programming (AOP) works on capturing and implementing crosscutting concerns while Model-driven architecture (MDA) provides standards to explicitly separate platform independent concerns from platform-specific ones. Therefore, the SoC principle is both applied to MDA and AOP, and these two techniques seem to be complementary to each other.

To support AOP in MDA, crosscutting concerns should be captured and modeled

at the different level at various level of abstraction while some model transformation is required to work with these models and to preserve their consistency.

Aspect-oriented languages and frameworks have been demonstrated in the literature together with an increasing number of tools in order to provide increased modularity and to support related quality attributes. Ideally, programming languages and modeling languages should be mutually supportive. However, crosscutting concerns cannot be completely captured and illustrated in modeling artifacts. Additionally, working with different models at different levels of abstraction and preserving their consistency requires well-defined and automated model transformations. These transformations help developers to reduce the effort of software maintenance activities such as reverse engineering and refactoring.

1.1 Objectives

Our first objective is to explicitly capture crosscutting concerns at the modeling level. The second objective is to provide a set of model transformations to map different models to each other and manipulate them. To meet these objectives, we set a number of goals: 1) To provide an extension to the UML metamodel to propose a model that is independent from any programming language and abstracted away from platform specific details. 2) To propose transformation patterns. 3) To illustrate how these patterns can be deployed to provide specifications of complex transformations.

1.2 Organization of the dissertation

The remainder of this dissertation is organized as follows: In Chapter 2 we provide the necessary background for this thesis. In Chapter 3, we discuss the problem and motivation behind this research and in Chapter 4 we discuss our proposal. We describe our methodology for modeling crosscutting concerns in Chapters 5 and 6. While the first discusses how to extend UML metamodel to support crosscutting concerns, the second describes a UML extension to model AspectC++ constructs by adding language-specific concepts to the language-independent UML extension. In Chapter 7, we introduce a set of model transformation patterns and describe how they can be deployed to provide model transformations to map different aspect models to each other. In Chapter 8, we describe three case studies to demonstrate how the proposed UML profile can be applied and how it can be used for modeling of complete AOP system. Furthermore, we demonstrate applying transformation to language-independent model to produce the language0specific one. In Chapter 9, we discuss related works and evaluate our approach. Finally, we list our conclusion and provide recommendations for further works in Chapter 10.

Chapter 2

Background

In this Chapter, we discuss the necessary background to this research, starting with aspect-oriented programming in Section 2.1, followed by UML and its extension techniques in Section 2.2. In Section 2.3, we describe model-driven architecture. Finally, model transformations is discussed in Section 2.4.

2.1 Aspect-oriented programming

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 [11]. Their implementation ends up cutting across the decomposition of the system. Aspect-orientation is a term used to describe approaches that explicitly capture, model and implement crosscutting concerns (or

aspects).

Examples of crosscutting concerns (or *aspects*) include persistence, authentication, synchronization and contract checking. Aspect-oriented Programming (AOP) [21] 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 [20], 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 new unit of modularity providing behavior to be inserted over functional components[1]. This behavior is defined in method-like blocks called *advice*. However, unlike a method, an advice block is never explicitly called. Instead, it is only implicitly invoked by an associated construct called a *pointcut expression*. 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

[1]This is not all that the AspectJ model provides. We focus on what is common among the AspectJ family of languages is a group of languages whose design dimensions have heavily influenced by AspectJ. This group includes AspectC++ and AspectC# among many others..

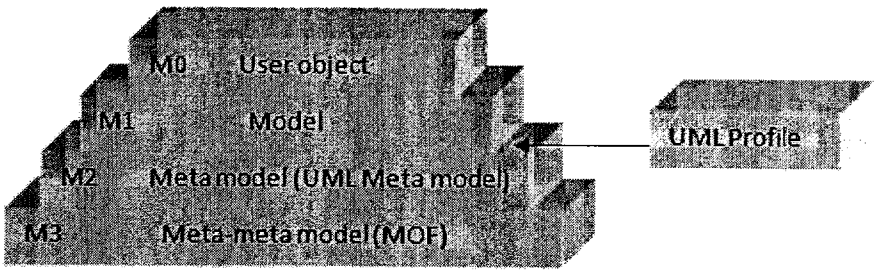

Figure 1: The meta object facility (MOF) 4-layered architecture.

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 [22].

2.2 UML and profiling

The Unified Modeling Language (UML) [45] is a de facto standardized modeling language maintained by the Object Management Group (OMG) that can be deployed (though not being confined) to represent object-oriented systems.

The UML is described by OMG as a 4-layered architecture shown in Figure 1.

- M3 is the meta-metamodel layer which is defined by the UML Meta Object Facility (MOF) [37]. The *metamodel* is a model that defines the structure, semantics, and constraints for a family of models [30]. MOF is a representation of the UML metamodel and it describes a small set of concepts (such as classes and packages) that allow one to define and manipulate models of the

metamodel. It enables metamodeling of UML-level metamodels, thus allowing new metamodels, and consequently new modeling languages, to be defined.

- M2 is the UML metamodel and it defines modeling languages such as UML. The UML metamodel is a representation of UML elements together with their interrelationships.

- M1 describes a user-defined UML model. All static or dynamic diagrams produced during software development or maintenance lie on this layer.

- M0 is the last layer or data layer and the real objects are describes in this layer.

The advantage of UML is categorized as follows:

- UML supports different degrees of precision, therefore it can be used at various level of abstraction for providing a lightweight, simple or very complex model.

- It provides different views of the same model that are mutually consistent.

- It proposes graphical representation that is easy to understand.

In the context of software maintenance, the UML supports reverse engineering of an object-oriented program through its transformation and representation at a higher level of abstraction. With the introduction of aspects to represent crosscutting concerns, there have been in the literature a number of approaches to support aspect modeling. These approaches are classified into two different main categories [3]:

1. Deploying standard UML in a way where existing elements (e.g. classes in a class diagram) can represent aspects.

2. Extending the current UML semantics and elements to provide explicit support for crosscutting concerns. For the latter approach, there are two ways for extending UML:

   (a) Manipulating the MOF: This is a heavyweight extension mechanism, and it tends to be currently impractical due to lack of tool support.

   (b) Building a UML profile [33]: The UML profile is an extension mechanism for building UML models that can be used to define domain-specific modeling languages (DSML). Using UML profiling mechanism, the produced DSML conforms to syntax and semantics of the UML; hence it can not violate any rules of UML.

   The UML profile is a subset of a UML metamodel that is defined using stereotypes, tagged values, and constraints for adapting UML meta elements to the constructs (in our case aspect-oriented constructs) of the new domain. As the UML profile does not define new elements for the UML metamodel, it can be considered as a lightweight extension mechanism. More specifically, UML profiles include the following:

   - Stereotypes to create new model elements for a specific domain.
   - Tagged values to define additional properties to model elements.
   - Constraints that add rules to restrict the way the model elements operate in the context of the new domain.

   An advantage of adopting a UML profile is that it allows modeling with

generic UML tools. Furthermore, developers can work with well-known UML notations and concepts, reducing the need to learn new modeling languages. Additionally, it is possible to combine stereotyped and non-stereotyped elements together for proposing a complete model of a software system.

2.3 Model-driven architecture

Models provide an abstraction of a software system by putting away irrelevant details while focusing on important concepts. Models help developers to deal with the complexity of a software system and understand it. Through models, developers, designers and stakeholders can talk about a system and reason about it. When talking about models in software engineering, we generally consider artifacts like class diagrams, collaboration diagrams, state diagrams and so on. Model-driven Architecture (MDA) [40, 30] is a model-centric approach proposed by the Object Management Group (OMG)[2]. In MDA, using well-defined notations, different models are proposed which capture different aspects of software systems. Furthermore, the OMG provides a conceptual framework for MDA, which contains a set of standards for expressing models, their relationships and model transformations [7]. MDA involves defining two different models at various levels of abstraction: the platform independent model (PIM) and the platform specific model (PSM). The PIM is a set of models of system

[2]http://www.omg.org/

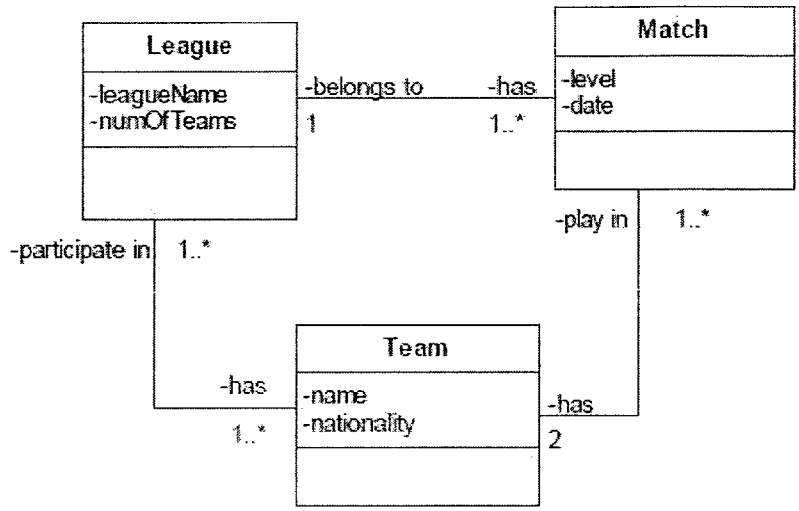

Figure 2: The PIM model.

functionalities that abstracts away technical details. It can be used to model concepts at the *analysis* level that represents the *core business logic* of the system as defined in methodologies like the Rational Unified Process (RUP) [25]. Figure 2 illustrates a class diagram that can be considered as a PIM. There is no constraint about specific platform in this diagram, hence it can be implemented by different programming languages.

In a MDA context, each constraint which is implied by the choice of programming language, hardware, operating system, communication networks and protocols is considered a platform. The PSM is a specification of the system that extends PIM by adding design constructs that are related to the specific platform. The PSMs might include several models, starting with high-level architectural models, followed by lower-level design models, all the way down to the executable code. Figure 3 is

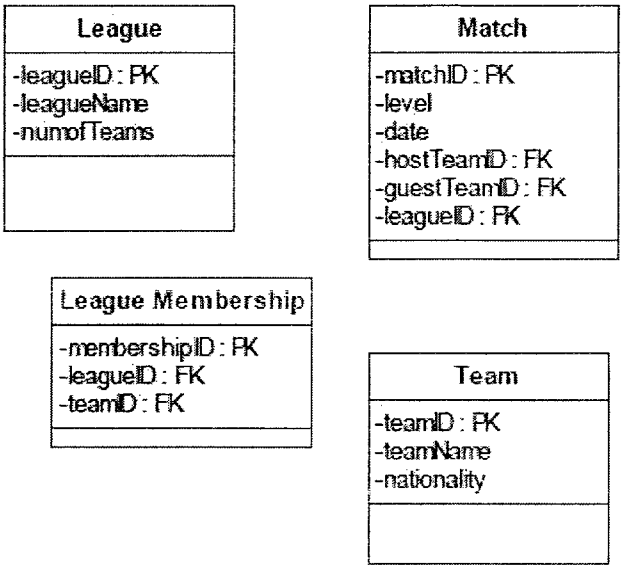

Figure 3: The PSM model

a relational database schema which is considered as a PSM model of the Figure 2. In this schema, some relational database elements such as primary keys are added to the PIM model for mapping the PIM model to relational database elements.

A typical MDA process is shown in Figure 4. First, a set of PIM models are proposed, after that a set of transformation is provided to obtain a PSM model by adding language-specific details to customize the PIM model. Finally, another set of transformations is applied to the PSM to obtain the final executable code.

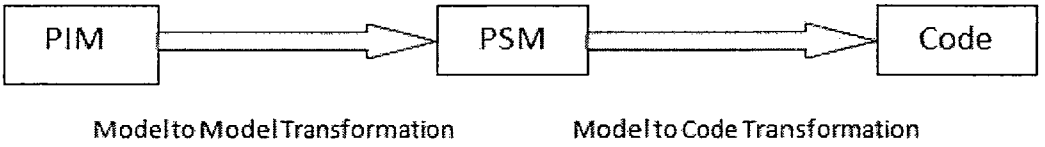

Figure 4: The MDA process starts with building the PIM model that is subsequently transformed into the PSM and finally to an executable code.

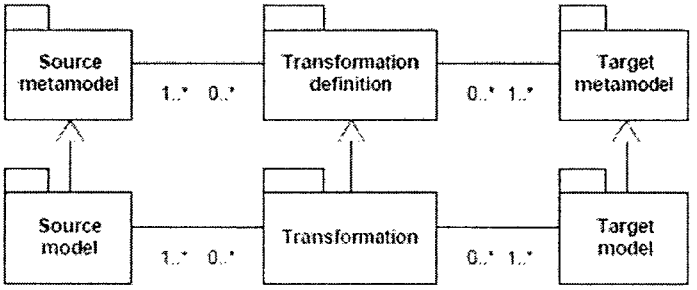

Figure 5: Transformation concepts in MDA.

2.4 Model transformations

A model transformation in MDA is an automated process that takes as input a model conforming to a given metamodel and generates as output another model conforming to a given metamodel. A model transformation generates a target model according to a set of rules that together describe how a model in the source language can be transformed into a model in the target language [23]. In Figure 5, we illustrate the transformation definition and its relation with the source and target metamodels.

For writing model transformations, it is necessary for a developer to have a clear

understanding of the abstract syntax and semantics of both the source and the target models. While metamodeling is the common approach for defining the abstract syntax of the models, its elements and their relationships, a model transformation language is required to provide automated, well-defined transformations.

2.4.1 Characteristics of a model transformation

In the following, we discuss characteristics of a desirable model transformation. According to [47], the desirable characteristics for a model transformation are:

1. Precondition: It should be possible to describe conditions under which a transformation can be applied.

2. Composition: It should be possible to combine different existing transformations to build a new composite one.

3. Form: The acceptance of a language transformation depends on the form that it used for defining a transformation.

4. Usability: It depends on the language and the developer's background.

These characteristics provide a measure to check the quality of model transformation languages and technologies. Czarnecki *et al.* [9] classify the existing model to model transformation approaches into:

- Direct manipulation approaches: it is the developers' responsibility to implement transformation rules from scratch using a programming language such as

Java. Therefore, this approach seems to be impractical.

- Relational approaches: it seems to provide the balance between flexibility and declarative expression. Relations in this approach don't have side-effects and this approach often supports backtracking.

- Graph-transformation-based approaches: this approach is based on heavily theoretical work in graph transformations. This approach is powerful and declarative but it is complex.

- Structure-driven approaches: it can be used in the context of certain kinds of applications such as generating Enterprise JavaBean (EJB) [3] implementations and database schemas from UML models.

- Hybrid approaches: the hybrid approaches combine different techniques from other approaches.

Czarnecki *et al.* [9] also discuss the applicability of these different approaches.

2.4.2 Query-View-Transformation(QVT) language

In this thesis, we use Query-View-Transformation(QVT) [34] that is proposed by the OMG as a hybrid approach to implement specifications of the model transformations for manipulating models of crosscutting concerns.

[3] http://www.oracle.com/technetwork/java/index-jsp-140203.html

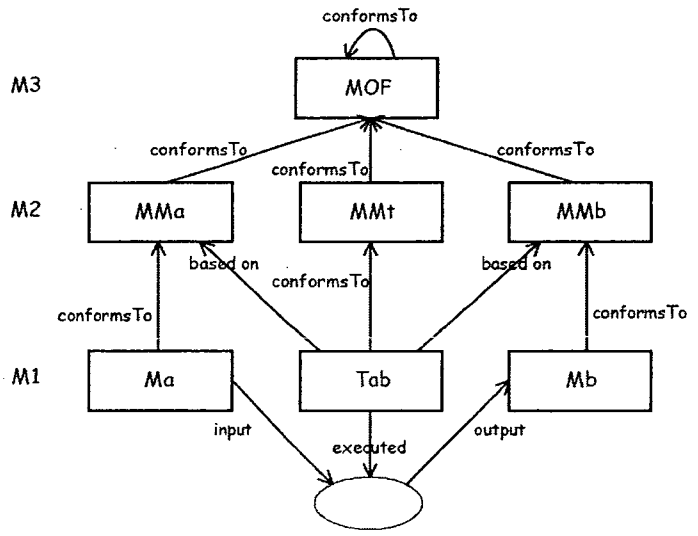

Figure 6: The QVT Operational Context.

The Query-View-Transformation language (QVT) is the OMG standard for defining model transformations that can be used not only for PIM-to-PSM transformations, but also for defining views on models and synchronization between models [26]. *Query* is a extended version of OCL 2.0 that is used to provide expressions evaluated over a model. *View* is a projection on a model that is completely derived from another model. Finally, *Transformation* is a process of automatic generation of a target model from a source model, according to a transformation definition [23, 38].

In Figure 6, the QVT operational context is presented. M_a and M_b models conform to MM_a and MM_b metamodel respectively. M_a is a source metamodel and M_b is a target metamodel. Furthermore, transformation T_{ab} conforms to MM_t metamodel and it is applied to map M_a to M_b. In addition, all metamodels(MM_a, MM_b and MM_t) are based on MOF.

The QVT metamodel defines three sublanguages: *Relations*, *Core* and *Operational*

Mapping for transforming models based on the Object Constraint Language (OCL). *Relations* is a declarative transformation language that specifies relations over model elements. *Core* is a declarative transformation language that simplifies the *Relation* language. *Operational Mapping* is an imperative transformation language that extends *Relations* with imperative constructs. In QVT, a transformation is specified in the form of mappings or relations. We choose QVT because:

- It is a standardized language that enjoys wide acceptance.
- There are some tools that support this language and it is possible to write, edit and execute QVT transformations easily.
- It is powerful enough to provide complete model transformations whilst its expressions are not complex.

In the next Chapter, we discuss the problem that has motivated this research.

Chapter 3

Problem and motivation

In this chapter, we discuss the problem and the motivation behind the research that constitutes the scope of this dissertation. The primary motivation behind this thesis is that combining MDA and AOP can increase the maintainability of the system because of a better separation of concerns. The MDA separates application logic from specific implementation technology and AOP modularizes crosscutting concerns into aspects; therefore, these two approaches can complement each other. Our objective is to raise the level of abstraction and provide a model for AOP system that can be adapted to any platform, either aspect-oriented platform or not. Wampler [54] discusses the main challenge that MDA deals with. The majority of applications are still developed by writing code using programming languages without any modeling. He also proposes several issues that MDA must address to fill the gap between the vision of MDA and the reality of the current software system development process. He concludes that modeling approaches should specify the software system completely and precisely.

Additionally, the model to model mapping must be suitable for automation and the PSMs models, including implementations, should be generated with minimal manual effort.

In this dissertation, we aim to:

1. Provide an extension to support modeling of crosscutting concerns at the PIM level completely leading to modularizing main concepts and aspects separately.

2. Provide model to model transformation to map a PIM model to a PSM model deploying QVT as a standardized transformation language.

Some of the existing approaches deal with modeling crosscutting concerns that deploy models that are language-specific. We are not discarding the language-specific approaches, but we use the best practices to provide a languages-independent model by abstracting away any language-specific parts. Additionally, language-specific modeling approaches can be used to provide language-specific models.

The motivation of this dissertation is to facilitate separation of pervasive features that are tangled with other system features and to support their transformation across different levels of abstraction by providing reusable model transformations in a standard format. We discuss our proposal to the above problem in the next chapter.

Chapter 4

Proposal

In this chapter, we discuss our research proposal. To provide support for crosscutting concerns in MDA, we propose to implement the following:

1. Modeling crosscutting concerns at the PIM level.
2. Modeling crosscutting concerns at the PSM level.
3. Providing transformation patterns as a general template for specifying model transformations.
4. Providing model to model transformation to map PIM to PSM.
5. Providing model to model transformation to map PSM to PIM.

In the literature, several UML extensions are proposed for modeling crosscutting concerns. Most of them rely on specific programming languages, specially AspectJ [12, 19, 39]. In some other works, the precise, well-defined notation and a

graphical representation that is supported by current case tools are either missing or incomplete. We propose an extension to the UML metamodel in order to model aspects. This extension is completely independent from any specific programming language and thus can be used to support generic aspect-oriented modeling. In Figure 7, the proposed methodology to support AOP in MDA is illustrated. In modeling part, we plan to define a domain model that constitutes the most important concepts that define a system as AOP. After that, we map these concepts to their correspondings in the UML metamodel to provide a UML extension. Finally, we propose this extension in different formats such as XMI that can be used by different CASE tools.

To provide the language-specific model, we add language specific constructs to the proposed generic model. In this dissertation, we provide a UML extension for the AspectC++ AOP language to illustrate how PSM modeling can be done.

Previous works either do not propose a transformation specifications using an standard language or they propose model transformation that are dependent on specific applications [41, 48]. Consequently, it is not possibel to use their transforation for manipulating crosscutting concerns' models. In this project, as illustrated in Figure 7, the model transformations are proposed to be used in forward engineering and backward engineering of the system. For providing model transformations, first, we propose a set of transformation patterns that can be used as a generic solution for writing specific model transformations. We use these patterns to propose well-defined transformations in a standardized language (QVT) to map the PIM model of aspects to the language-specific one for forward engineering. In this activity, the input of the

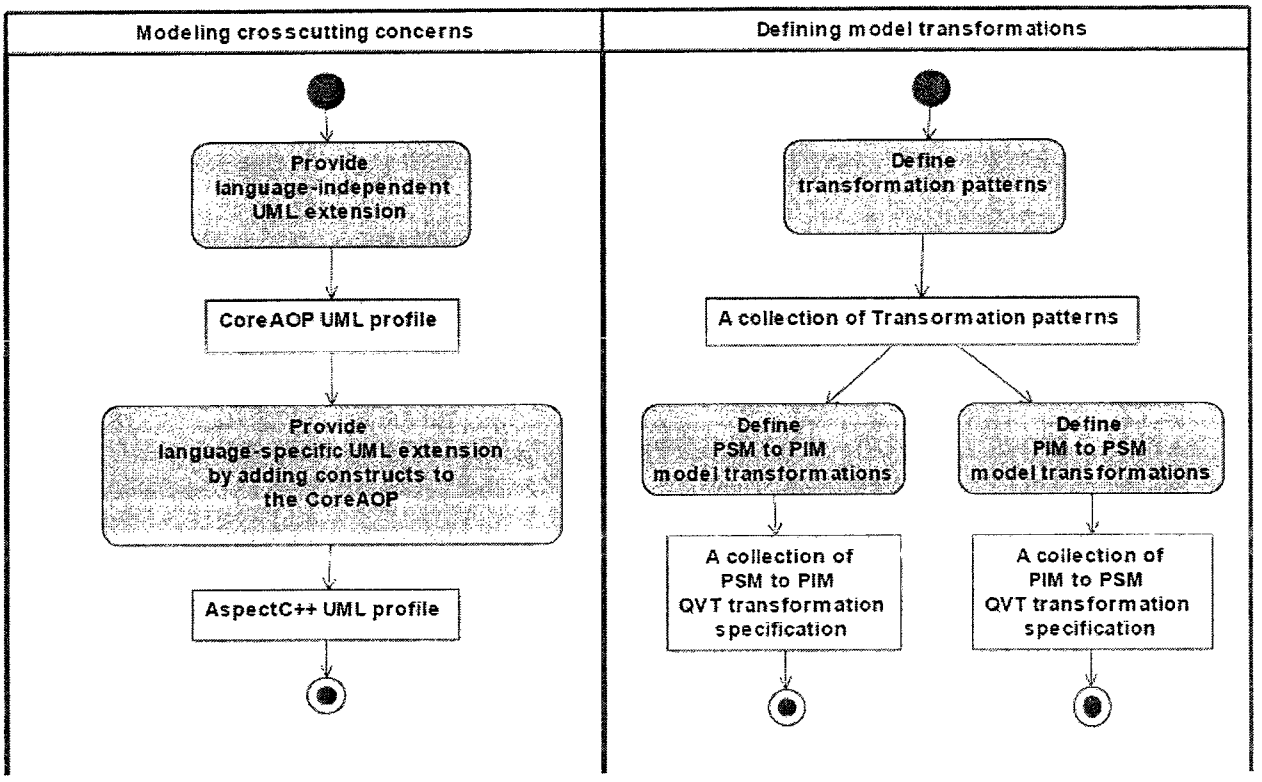

Figure 7: The proposed methodolgy to support aspect-oriented programming in model-driven architecture.

model transformation is a model based on a language-independent UML extension. It is also possible to use the proposed pattern to define reversible transformations to map a PSM model to a PIM model for reverse engineering of the system. Finally, we will demonstrate how to use a current CASE tool, namely Eclipse EMF [8], to automate the process of modeling and writing transformations. In order to demonstrate our approach in a practical situation, three different case studies are deployed.

4.1 Expected contributions and benefits

The expected contributions of our proposal are to provide support during the modeling and design phase of the software life cycle. Potential beneficiaries of this approach include system developers who can provide consistent, reusable and maintainable artifacts of the software system. Modularization of crosscutting concerns at early stages of the software development process leads to enhanced separation of concerns. In the following chapters, we discuss our methodology to provide a complete, precise UML extension to model crosscutting concerns. Additionally, we explain how we propose a language-specific UML extension by extending the generic model to support AspectC++. Finally, we define model transformation patterns and illustrate how they can be used as a template for providing PIM to PSM and reversible transformations.

Chapter 5

Modeling crosscutting concerns

5.1 Introduction

Approaches in the literature to model crosscutting concerns tend to be language-specific. At the modeling level, the reception of AOP has long been focused on the modeling of AspectJ programs, and there exists no model that is generic enough to capture non-AspectJ aspects either as a source language during forward engineering or as a target language during reverse engineering. Our objective is to provide such modeling in a language-independent manner. To achieve this objective, we need to specify a core model for aspectual representations. In particular, we present our proposal for a UML profile that models crosscutting concerns independently from any technology such as a specific programming language.

The remainder of this chapter is organized as follows: In Section 5.2 we present our methodology by introducing a UML profile for modeling crosscutting concerns.

Finally, possible applications for aspect-oriented language-independent profile are proposed in Section 5.3.

5.2 The CoreAOP UML profile

In this thesis, we propose a UML-based profile that is built on Level 2 of the 4-layered architecture shown in Figure 1. We call this profile *CoreAOP* because it can be used to model crosscutting concerns completely independent from any platform. In the following the CoreAOP profile specification is presented and the description of the stereotypes are illustrated in Figure 8.

Profile Name: CoreAOP

Version: 1.0

Reference Meta-model: UML meta-model

Description: The CoreAOP profile extends the UML metamodel to explicitly capture crosscutting concerns.

Based on Selic [46] that describes a systematic method for defining UML profiles, our methodology for proposing a UML profile to support crosscutting concerns includes the four steps shown in Figure 9.

Name	Base Meta-Class	Semantic	Related constraints
<< Aspect >>	Class	An aspect encapsulates static and dynamic features of a crosscutting concern.	
<< Advice >>	Behavioral Feature	An advice block encapsulates behavior. Whenever the specific pint of the program execution is reached, an advice is activated by an aspect.	1. Only classes that are stereotyped as aspects can have behavioral features that are stereotyped as advice. 2. The name of the advice is similar to the name of the attached pointcut.
<< Pointcut >>	Operation	Pointcut specifies a set of join points by determining on which condition an aspect shall take effect.	1. Only classes that are stereotyped as aspects can have behavioral features that are stereotyped as pointcut. 2. It is possible to have unnamed pointcut.
<< Joinpoint >>	UML collaboration diagram	Join point is a specific point in the control flow of a program.	

Figure 8: The CoreAOP Stereotypes Specification.

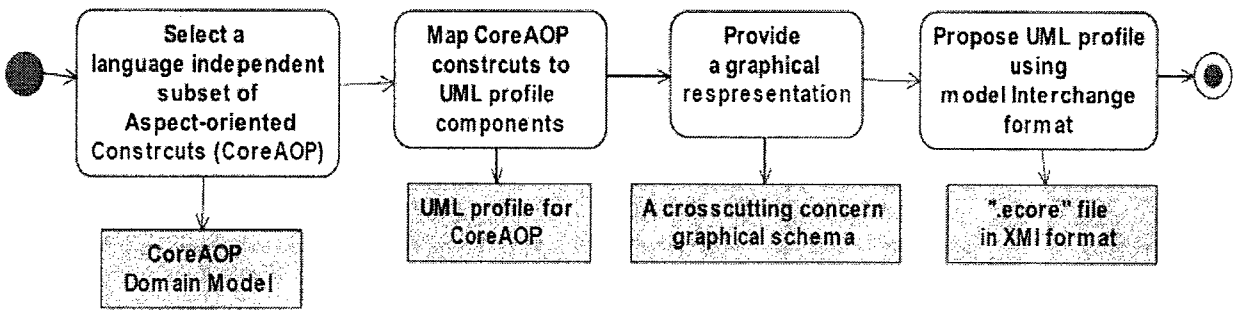

Figure 9: The methodology for creating a UML profile for crosscutting concerns.

5.2.1 Defining a domain model

The domain model is the conceptual model of the system that specifies all constructs that need to be represented. It describes the system scope and can be used as the domain vocabulary of the system. To describe the elements of the domain model for an AOP system, we need to go back to the first principles and discuss what characteristics make a system AOP. Are there core concepts which are necessary and sufficient to qualify a system as AOP? In an article that gained high popularity, authors Filman and Friedman [11] describe two properties that are essential for AOP: *Quantification* and *obliviousness*. Quantification implies that one should be able to execute statements of the form *"In program P, whenever condition C occurs, execute action A." Obliviousness* implies that components should not necessarily be built under the consideration that some aspectual behavior will be applied on them, i.e. they should maintain no visibility over aspects. As the AspectJ programming language has influenced the design dimensions of other general-purpose aspect-oriented languages, it has, in essence, dictated a collection of implementation concepts to support AOP. Along the lines of the AspectJ model, we define a domain model with the following elements: (See Figure 10)

- A set of language constructs that are core concepts in our domain. We choose aspect, join point, pointcut and advice as a subset of aspect-oriented constructs which we refer to as CoreAOP model. The CoreAOP model contains the constructs that conceptually introduce AOP [11, 14, 50].

- A set of valid relationships between the CoreAOP concepts.
  - *Has a precedence of*: The precedence relationship defines a relationship between two aspects, which is used to determine the execution order of advice block if more than one aspect affects the same join point.
  - *Has between Aspect and Advice*: The aspect contains zero or more advice.
  - *Has between Aspect and Pointcut*: The aspect contains zero or more pointcuts.
  - *Association between Pointcut and Advice*: Each advice is applied to one specific pointcut. Whenever the specific points in the execution of the program are reached, the advice is triggered and is executed before, after or instead of it. Different advice can be triggered based on one pointcut, therefore the multiplicity for the association end of advice is one or more.
  - *Aggregation between Pointcut and Joinpoint*: Pointcut is a predicate that matches join points and one joinpoint can be used in different pointcuts, therefore aggregation is used to illustrate the relationship between pointcut and joinpoint.
  - *Self aggregation for pointcut*: It is possible for a pointcut to be composed of other pointcuts.
- A set of constraints.
- A concrete syntax or graphical representation of the domain model.

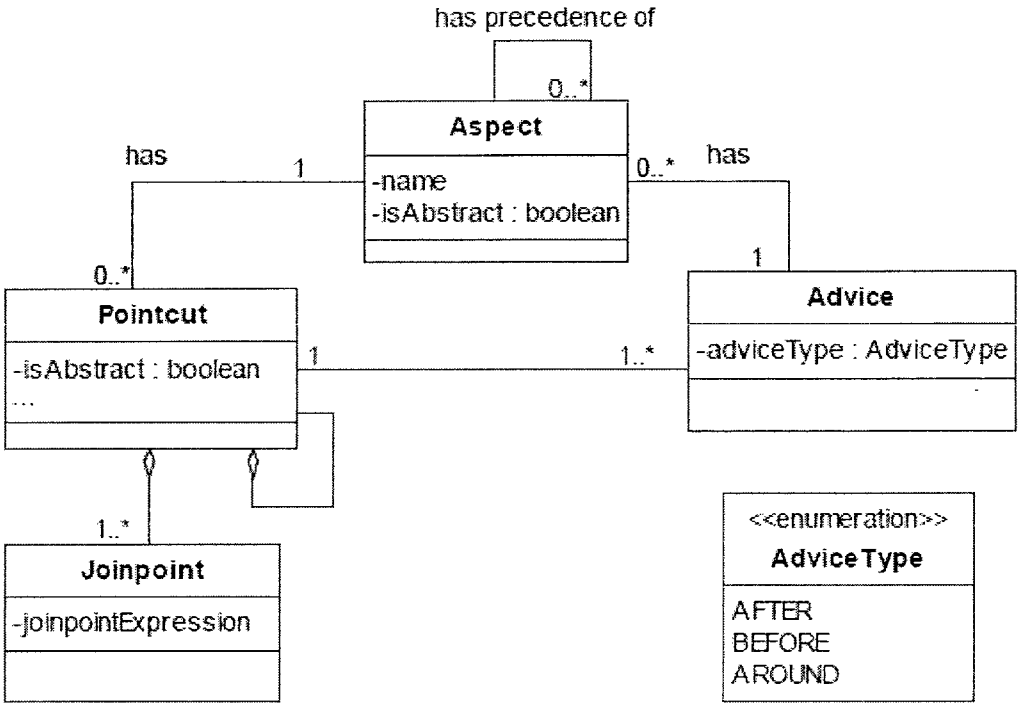

Figure 10: The CoreAOP domain model for crosscutting concerns.

- Semantics of the domain model representation.

5.2.2 Mapping the domain model to the UML metamodel

To map the domain model to the UML metamodel, we need to identify the most suitable UML metamodel base concepts for each element in the domain model. Each element of the domain model of Figure 10 must be individually mapped. The mapping is discussed subsequently:

1. **Aspect**:

   Semantics: An aspect definition encapsulates static and dynamic features.

Additionally, much like a class, a concrete aspect can support inheritance whereas an abstract aspect can enforce inheritance[1]. Therefore, the UML metamodel `Class` is a good candidate for representing an aspect. This is because the `Class` metaclass is a classifier and subsequently a generalizable model element in the UML metamodel. Being a classifier, it enforces the extended element to have a name and if it is inherited from a generalizable element it can be abstract or can be used as a base class for extension.

Attributes: Aspect contains the following attributes:

- **name:EString**: Aspect inherits the features and relationships of `Class`, therefore it has a `name` attribute.
- **isAbstract:EBoolean**.

2. **Advice**:

Semantics: An advice block encapsulates behavior. It indicates what happens whenever the program reaches specific points during its execution. In the proposed profile, we model advice as a stereotyped `Behavioral feature`. In the UML class diagram metamodel, the `Behavioral feature` can be either a `Method` or an `Operation`. An advice is never invoked explicitly and it does not have any parameters. Therefore, it cannot be modeled as a method. In addition, as discussed in [27], "a UML operation is a declaration with name and parameters" and as such it is an

[1]Much like in OOP, in AOP we can also have different forms of inheritance, such as for extension, or for specification.

abstract definition without implementation. Therefore, semantically, an advice is not an `Operation` because it is not only the declaration but it also contains the implementation. In Figure 11 we show how to extend the metaclass `Behavioral feature` for modeling advice. Therefore, an advice is a stereotyped `Behavioral feature`. In the UML behavioral package, collaboration and state-chart diagrams are included in meta-class `Behavioral feature`. Therefore, they can be attached to advice as a `Behavioral feature` class and used to specify advice implementation.

Attributes: An advice contains the following attributes:

- **name:EString**: An advice is a model element in the UML metamodel therefore it has a name.
- **adviceType:AdviceType**: An advice has different types, and it can be executed after, before or instead of specific events defined in pointcut expressions. Consequently we can add an enumeration type that contains different types of advice, and call it `adviceType`.

3. **Join point:**

Semantics: Each join point specifies a well-defined place during the execution of the program where the aspect interacts with the core functionality. Along the lines of the work described in Fuentes *et al.* [15], we deploy sequence diagrams for modeling join points where messages are stereotyped as join points. Sequence diagrams provide a graphical representation for

displaying join points that is simple to understand. Wildcards can be used for addressing classes and methods names. Here we have a star (*) representing any sequence of characters and a double-dot (..) representing any sequence of arguments.

Attributes: A join point contains the following attributes:

- **expr:EString** contains join point expression in String format.

4. **Pointcut:**

Semantics: A pointcut is a predicate of join points. A pointcut can include a name and as a result it can be easily reused (for example, to be attached to various advice blocks which provide different behavior for it). We therefore modeled a pointcut as a stereotyped operation. An operation has no body and it can be abstract. Semantically and conceptually it can be a good choice for modeling pointcuts. Moreover, there is an advantage to modeling pointcuts and join points with definitions that are independent from advice as this decoupling can promote reuse of the former.

Attributes: A pointcut contains the following attributes:

- **isAbstract:EBoolean** illustrates whether the pointcut is abstract or not.
- **name:EString**: If the pointcut is unnamed, we call it unnamed in this work.

In the UML metamodel, `BehavioralFeature` is owned by `Class`. Due to the fact that `Aspect` is an extension of the `Class` meta-class, and `Advice` and `Pointcut` are extensions of the `BehavioralFeature` meta-class, `Aspect` already contains `Advice` and `Pointcut` because of the association that exists between the metaclasses `Class` and `BehavioralFeature` in the UML metamodel. In other words, semantically, there is no need to explicitly create an association that relates an aspect to its advice and pointcuts. We impose two constraints during the definition of `Advice` and `Pointcut`.

Additionally, only classes that are stereotyped as aspects can have behavioral features that are stereotyped as advice or pointcuts. With appropriate constraints defined in the Object Constraint Language (OCL) [36], we confine advice and pointcuts to aspects.

In Figure 11, we illustrate the UML profile for modeling crosscutting concerns. Highlighted items are added to the UML class diagram metamodel for proposing a UML profile for modeling crosscutting concerns.

5.2.3 Providing a graphical representation

- **Aspect notation:** Like class, aspect is illustrated in folded or unfolded form that contains one additional compartment for pointcuts and advices. Using stereotyping, a new model element, `Aspect`, is added to the UML class diagram to represent aspect.

- **Advice notation:** `Aspect` has a behavioral feature that is stereotyped as

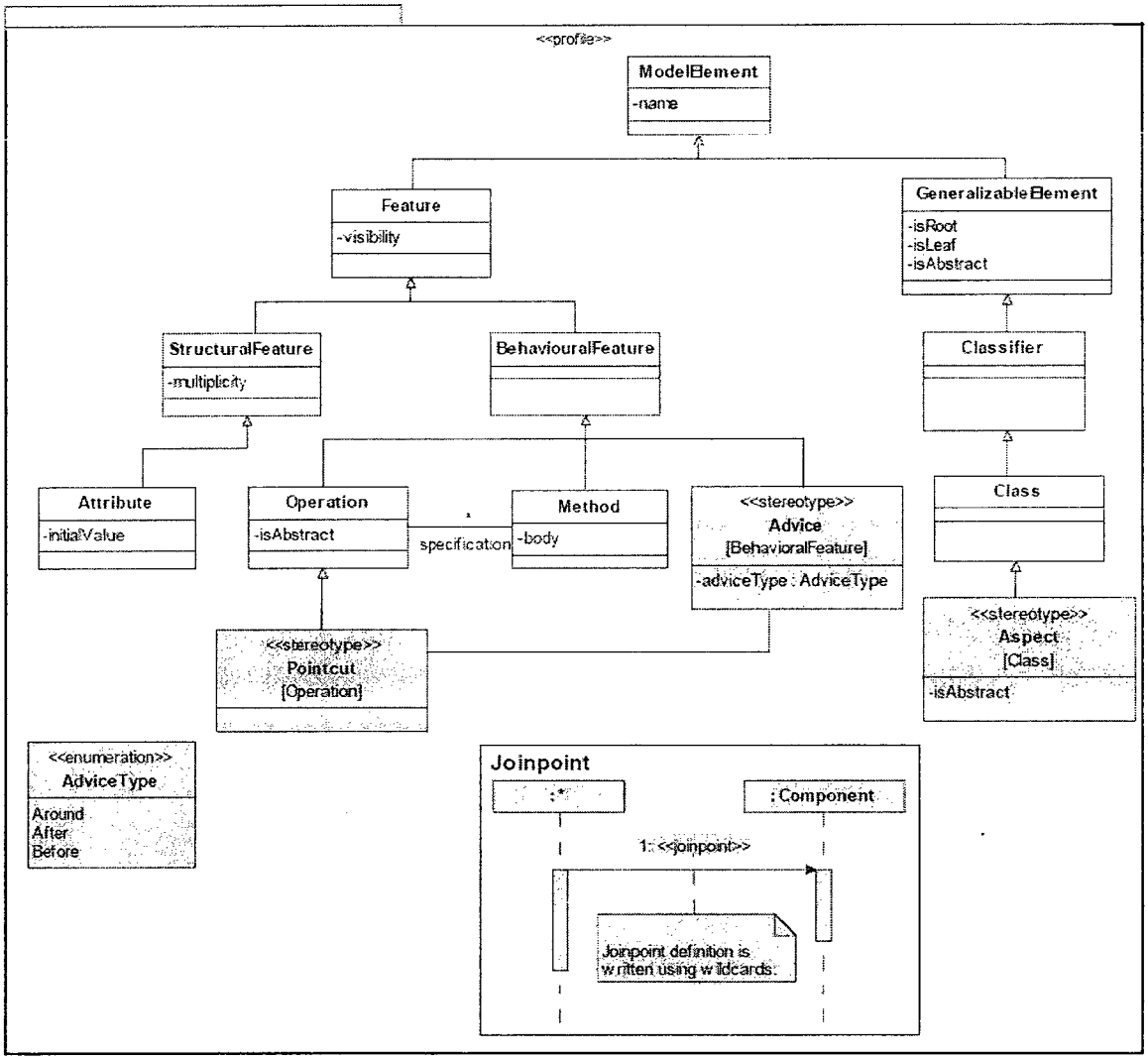

Figure 11: The proposed UML profile to support crosscutting concerns.

```
Context Aspect
inv:self.advices -> forall( a:Advice: | self.pointcuts -> select (name = a.name))
```

Figure 12: The OCL constraint to enforce the name property of the Advice in the Aspect Context.

```
Context Advice
inv:self.advices.pointcuts -> select ( name = self.name )
```

Figure 13: The OCL constraint to enforce the name property of the Advice in the Advice Context.

`Advice`. An advice is defined by its name and its `AdviceType` that can be `After`, `Before` or `Around`. As advice is declared without name and every element in class diagram notation should have a name; we use the name of attached pointcut. An OCL invariant constraint is defined for the name of the advice to enforce this property. This constraint can be written in two different forms that are semantically equal. The first one as illustrated in Figure 12 is defined over aspect.

This constraint indicates that for every advice that is stored in a list called advices in the body of an Aspect, there is a pointcut whose name is similar to the name of that advice. The second form is defined over Advice and illustrated in Figure 13.

- **Pointcut notation:** Every pointcut is an instance of the meta-class `Pointcut`. Aspect has an operation feature that is stereotyped as `Pointcut` and shows the parameter list that contains the related join points name.

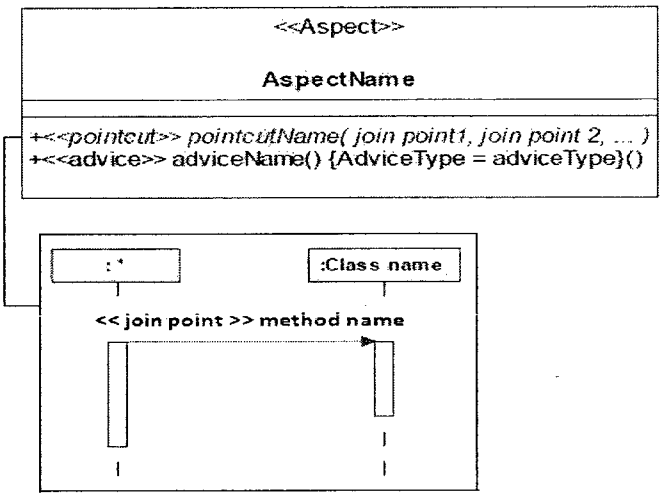

Figure 14: A graphical representation for modeling crosscutting concerns.

- **Join point notation:** A join point is defined in a sequence diagram that has a name and a specific message that is stereotyped as a `join point`. One or more sequence diagrams displaying join points are attached to a specific pointcut.

Figure 14 illustrates how to model crosscutting concerns in UML tools that supports our UML profile.This representation hides unnecessary details and represents crosscutting concerns in a manner where there is no need for any additional (e.g. textual) specification. Thus the produced model can be manipulated or verified automatically by any tool that works on UML diagrams. Using this model, crosscutting concerns and their relationships with other model elements can be displayed.

5.2.4 The proposed profile using a model interchange format

To deploy the profile in available CASE tools such as Eclipse Modeling Framework (EMF) [8], it is necessary to provide a persistent interchange format. St-Denis *et al.* [49] define a list of requirements for model interchange formats (RSF [55], RDF [28], XMI [35], etc.) and discuss their advantages and disadvantages. We have decided to adopt XMI (XML Metadata Interchange). First of all it has wide industry and tool support. Furthermore, XMI is a metamodel to describe model elements and therefore it is completely compatible with UML. Additionally, XMI uses XML syntax and therefore has all the advantages of XML.

The XMI format allows one to capture our metamodel in a specific, formal, persistent form that is required for defining model to model transformations [47]. In the EMF framework that is a Java framework and code generation tool to work with standard models, each metamodel is represented as an `.ecore` file in XMI format. We have created the `CoreAspect.ecore` file that contains all the CoreAOP constructs, so it can be used as a metamodel for any aspect-oriented model.

There are several ways of getting a UML model into XMI form:

1. Using an XML or text editor to create the XMI document directly. This is the most direct one to illustrate a model in XMI format.

2. Using XML schema definition (XSD) [13].

3. Using EMF

(a) Export the XMI document from a modeling tool such as IBM Rational Rose [2] in EMF.

(b) Using Java annotation [3] to annotate Java interfaces to produce a XMI file.

We are going to show how a XMI model of CoreAOP profile is generated from Figure 11 using EMF. For modeling the CoreAOP profile, its elements are mapped to the EMF classes. The complete class hierarchy of the Ecore model is illustrated in Figure 15. The CoreAOP profile in XMI format is illustrated in Figure 16 in the tree-based display model.

- Create an EMF project.
- Create an Ecore model that is called `CoreAspect.ecore`. The first element in this model is `EPackage` that is called `CoreAspect`. The following steps illustrate how to add other model elements to this package.
- Add a new child to the package that can be:
  - EAnnotation
  - EConstraint
  - EClass: It is used to illustrate the constructs in metamodel and it has an attribute called name. It also can have zero or more attributes of the following type:

[2] http://www-01.ibm.com/software/awdtools/developer/rose/

[3] JDK 5.0 Developer's Guide: Annotations. Sun Microsystems. 2007-12-18. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. (Retrieved 2008-03-05)

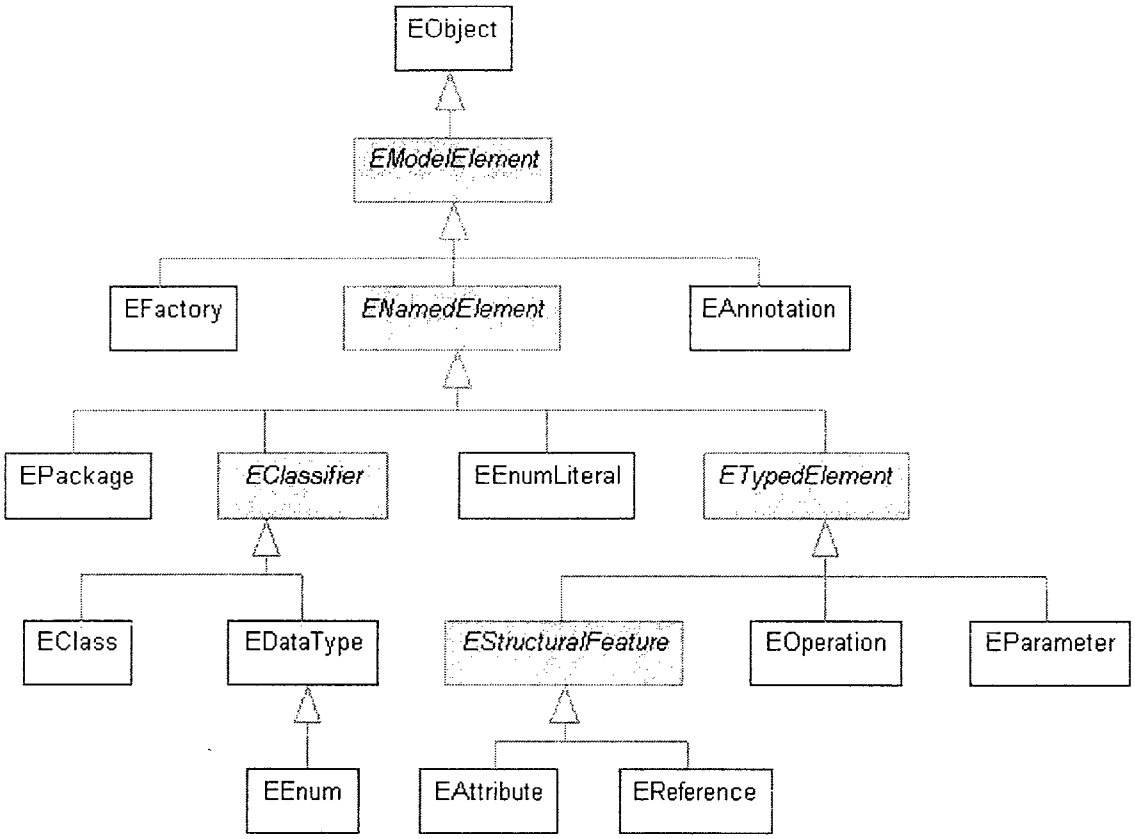

Figure 15: The complete class hierarchy of the Ecore model.

1. EAnnotation.
2. EOperation.
3. EAttribute: It is used to provide Class attributes. For example, `EClass` UML model element has `EAttribute` name of a type of `EString`.
4. EReferences: These references can be used to illustrate the association between two classes. The list of these references are described as follows.
   * **owner:Aspect** in `Advice` ECLass is a reference to an aspect to illustrate the association between aspect and advice.
   * **joipoints:Joinpoints** in `Pointcut` EClass is a set of join points that a pointcut predicate over them.
   * **owner:Aspect** in `Pointcut` EClass specifies the aspect that contains this pointcut.
   * **advices:Advices** in `Aspect` EClass is a set of advice in specific aspect.
   * **pointcuts:Pointcuts** in `Aspect` EClass. An aspect may contain a set of pointcuts.
   * **ownerPackage:Package** in `Aspect` EClass. Each aspect belongs to a specific package that encapsulate the crosscutting concepts.

In this profile, we have the following classes: `UMLModelElement`, `GeneralizableElement`, `Classifier`, `Class`, `BehavioralFeature`, `Operation`, `Aspect`, `CorePointcut`,

`CoreAdvice` and `Joinpoint`.

- EDateType: It represents the type of an attribute for example `int`, `float` or `java.util.Date`.
- EEnum: `AdviceType` is the enumeration type that specifies the type of Advice that can be before, after and around.
- EPackage

5.3 Discussion

Several applications can be proposed for an aspect-oriented language-independent profile. First, the CoreAOP UML profile is used to provide language-independent models of crosscutting concerns. Therefore, it is possible to model one crosscutting concern and present it in two different format: a graphical representation using UML and an XMI representation.

Second, due to the fact that we selected only the essential aspect-oriented language constructs, the most important characteristic of our UML extension is the language independence property, which can be used to provide a language-specific profile. By adding language-specific constrcuts, we extend the CoreAOP profile to provide model for specific AOP platform. In the following chapter, we describe how an AspectC++ profile is proposed for modeling AspectC++ systems.

Third, the language-independency makes it suitable for providing proper model

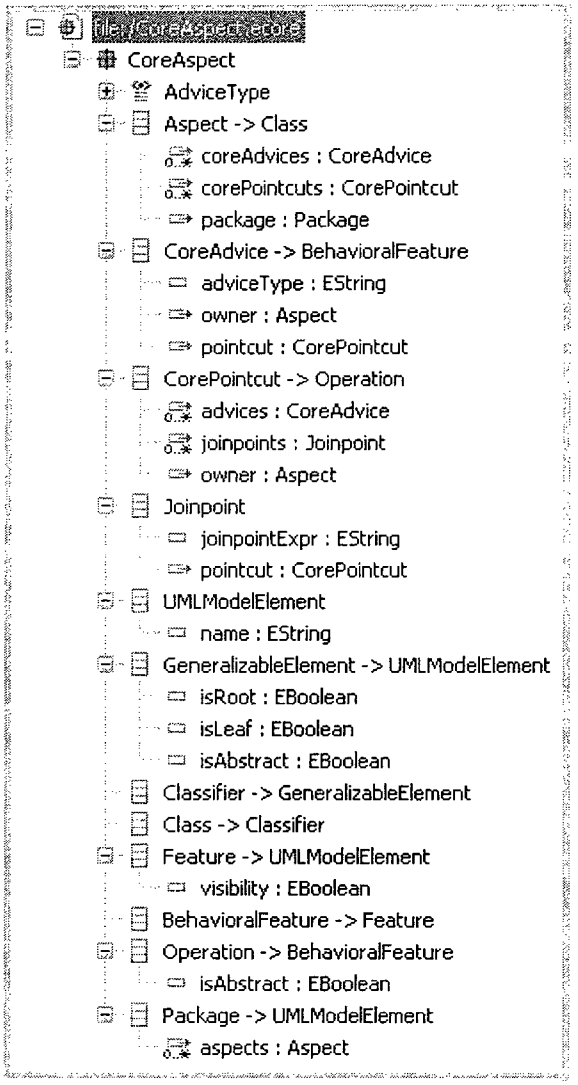

Figure 16: The CoreAOP profile in XMI format.

transformations for maintenance activities such as reverse engineering, forward engineering, language migration and reengineering. More discussion is provided in Chapter 7 and Chapter 8.

Additionally, if we have a model that is based on a specific aspect-oriented programming language (e.g., the AspectJ profile discussed in [12]), we can abstract away the details that are specific to the language and preserve the main concepts to accomplish reverse engineering.

Fourth, the EMF generator can create a corresponding set of Java implementation classes automatically from the model and the developer can edit these generated classes to add methods and instance variables.

Chapter 6

A UML profile for the AspectC++ programming language

6.1 The AspectC++ UML profile

The main important characterisitc of the proposed UML profile, *CoreAOP*, is the language independence property, which can be used to provide metamodels for other aspect-oriented programming language. Due to wide-popularity and tool support, there are several UML profiles for AspectJ but there is no UML extension to support modeling AspectC++ concepts. In this chapter, we explain how we propose a UML profile for modeling aspects based on the AspectC++ specification presented in [53]. To extend the CoreAOP metamodel to propose a new metamodel for a specific programming language, we need to identify the most suitable CoreAOP metamodel elements and individually map each element in the AspectC++ to their corresponding

element.

Profile Name: AspectC++

Version: 1.0

Reference Meta-model: CoreAOP UML Profile

Description: The AspectC++ profile extends the CoreAOP UML extension to explicitly capture crosscutting concerns in AspectC++.

AspectC++ is an aspect-oriented extension to the C++ language. The following paragraphs describe the elements in AspectC++ profile and how they map into CoreAOP constructs.

- *Pointcut* is the most important element of AspectC++ which contains a set of joinpoints. There are two types of pointcut:

  1. *Code pointcut*: that can be either *Execution* or *Call.*
  2. *Name pointcut*: it contains a set of entities such as *Type*, *Attribute*, *function*, *variable* and *namespace.*

  Pointcut in AspectC++ profile is the subclass of the `pointcut` in the CoreAspect Profile.

- *Joinpoint* declares a condition in which the aspect comes into effect. There are different types of joinpoints: *Code join points* are used to form code pointcuts

and *name join points* are used to form name pointcuts. Joinpoint is modeled in the way that we model joinpoint in CoreAOP profile.

- *Slice* is a piece of a C++ language code element that defines a scope and can be used by advice to extend the static structure of the programm. Due to this fact that there is no construct as `slice` in CoreAOP product, it is modeled as `StructuralFeature`.

- *Advice* in AspectC++ profile is the subclass of the `advice` in the CoreAspect Profile. There are 2 different types of advice in the AspectC++ specification:

  1. *Advice Code* it is bound to *Code joinpoint* and it is an action that is activated whenever the corresponding joinpoint is reached. This *Advice Code* shall be activated before, after or instead of a specific part of the program.

  2. *Introduction* is the second type of advice supported by AspectC++ that is used to extend program code and data structures.

- *Aspect* is the element of AspectC++ to collect advice, pointcut and joinpoints for implementing a common crosscutting concern in a modular way. We model aspect as a subclass for `aspect` in CoreAOP profile.

In Figure 17, we illustrate the UML profile for modeling crosscutting concerns in AspectC++. Highlighted items are added to the UML class diagram of the CoreAOP metamodel for proposing a UML profile for modeling AspectC++ constructs.

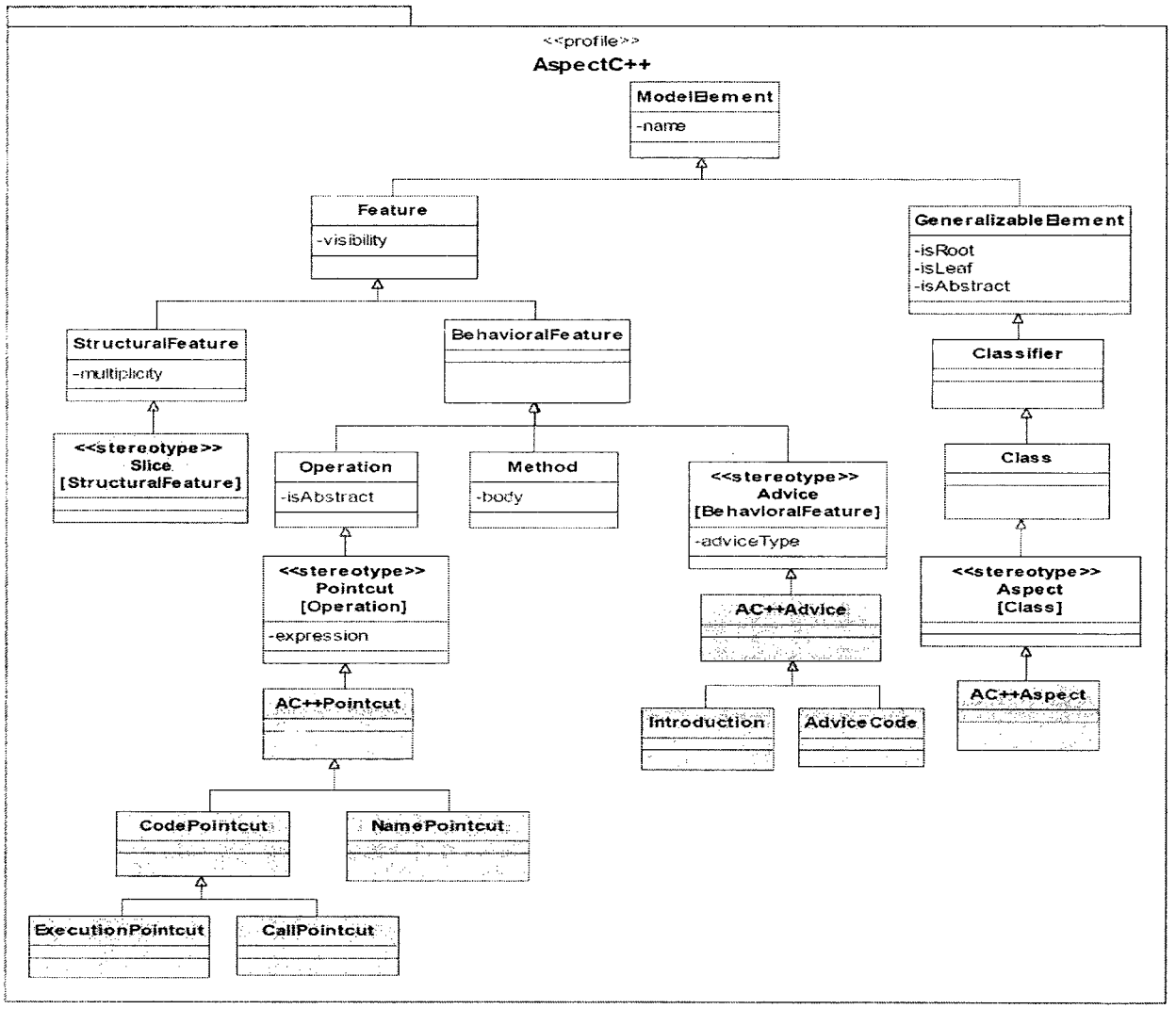

Figure 17: The proposed UML profile to support crosscutting concerns in AspectC++.

We extend the language-independent CoreAOP profile to provide a language-specific profile for AspectC++. The graphical respresentation that is proposed for illustrating aspects and its constructs in CoreAOP, 5.2.3 can be used to display AspectC++ aspects as well. The AspectC++ UML profile is an exemplification to show that the CoreAOP metamodel is pragmatic and enables implementation of the metamodel for aspect-oriented programming languages.

Chapter 7

Model transformations

7.1 Introduction

Dealing with different models and applying changes to all of them while preserving their consistency requires model transformations. In this Chapter, we explain how to specify QVT model transformations to manipulate aspect models.

Because writing transformations is tedious and time-consuming, there is a reusable solution to a general model transformation problem that is called transformation pattern [17]. In this work, we propose a set of transformation patterns that can be used as a template for specifying reusable model transformations to manipulate models in MDA.

The remainder of this chapter is organized as follows: In Section 7.2, we describe metamodels that are used as target or source metamodels for executing transformations. Transformation patterns are proposed in Section 7.3. In Section 7.4, we

present PIM to PSM transformation. Moreover, PSM to PIM model transformations are presented in Section 7.5. Finally, the applicability of our work is explained in Section 7.6.

7.2 Metamodels

We aim to provide model to model transformation to transform the PIM model of crosscutting concerns into PSM models and vice versa. Metamodeling is the best approach for providing source and target models. We use the CoreAOP UML profile to model crosscutting concerns in a language-independent form. This profile is completely described in Chapter 5. At the PSM level, we use the AspectC++ profile proposed in Chapter 6. In the following, we discuss a language-specific profile for AspectJ.

7.2.1 AspectJ metamodel

The AspectJ metamodel that we used in this project is proposed by [12]. This profile is a UML profile for modeling all AspectJ concepts. The following paragraphs present the UML metamodel for each AspectJ constructs.

1. **CrosscuttingConcern** meta-class relates aspects together and extends the meta-class `Package`.

2. **Aspect**'s characteristics are close to the features of the UML `class` therefore, `Aspect` extends meta-class `Class`.

3. **Advice** is the dynamic feature of aspect that is modeled as the meta-class `BehavioralFeature`. `AdviceExecution` is an attribute of the `Advice` meta-class and modeled by the enumeration `AdviceExecutionType`.

4. **Pointcut** extends the UML metaclass `StructuralFeature`. The `Pointcut` is an abstract meta-class in this profile and several different classes implement it. It is also possible to have combinations of pointcuts using `CompositePointcut`. In `CompositePointcut`, two or more pointcuts are combined using logical operators defined in `PointcutCompositionType`. This apsectJ profile does not have join points and there are several non-abstract subclasses for modeling pointcuts and join points together. It's a programmer's choice to select what kind of pointcut is needed. The different types of pointcut in AspectJ profile are illustrated in Figure 24.

7.3 Provide transformation patterns

Since the publication of Design Patterns by Gamma *et al.* [16], patterns are well known in software engineering. Patterns describe which problems software engineers can encounter, the context in which such problems may appear, and a general solution to them. In this thesis, we extend the notion of pattern and provide two transformation patterns as a reusable solution for specifying model to model transfomrations for models of crosscutting concerns.

For the provision of transformations for crosscutting concerns modeled using a

```
XYMapping {
  nm: String;
  enforce domain left x: X {
        context = c1 : XContext {},
        name = nm };

  enforce domain right y: Y {
        context = c2 : YContext {},
        name = nm };

  when {
        ContextMapping(c1,c2);
  }
}
```

Figure 18: The mapping pattern.

generic metamodel to AOP language-specific model, we deploy two patterns that are proposed in [17]. The first one is the *mapping pattern* used to establish one to one relations between elements from the source metamodel to the target metamodel. The second one is the *node refinement* pattern for obtaining a more detailed model from the abstract one. The mapping pattern is illustrated in Figure 18 while the specification of the node refinement pattern is presented in [17].

These transformation patterns are written using QVT *Relations* language. In order to underestand these patterns and how we use them to provide new transformations for crosscutting concerns, we briefly discuss the *QVT Relations* language below.

A Transformation written in the *QVT Relations* language consists of several relations that should hold between elements of the source and the target metamodels. The `when` and `where` clauses are used to constrain the transformations rules. The

`when` specifies the preconditions. For establishing the relation between the source and the target metamodel, all conditions specified in this clause should be evaluated to true. The postconditions are represented in the `where` clause. Once the relation is established, the condition in the `where` part should be satisfied and evaluated to true.

In the *Relations* language, domains are marked as `enforce` or `checkonly`. The `checkonly` indicates that the domain elements are read-only and cannot be changed after executing the transformation. While `enforce` indicates that after executing the transformation, the transformation engine should change the elements of the domain to satisfy the constraints that are proposed in the `where` clause.

In this dissertation, we provide two transformation patterns:

1. `Generic AOP node mapping pattern`: this pattern illustrates how one model element in one AOP metamodel can be mapped to another elements in another AOP metamodel at the same or various level of abstraction.

2. `CoreAOP model - AOP mapping pattern`: this pattern is a specialized verion of the previous pattern and it illustrate how it is possible to transform models that are provided using CoreAOP profile to another AOP models and vice versa.

7.3.1 Generic AOP node mapping pattern

- **Name:** The GenericAOPNodeMapping pattern.

- **Goal:** Provide relations between elements in the source metamodel to elements

```
enforce domain aOPLangMM nodeT1:NodeTarget1
                          {nodeT1Attr1 = attr1, nodeT2Attr2 = attr2};
```

Figure 19: The `Enforce` part of the `GenericAOPNodeMapping` pattern.

in the target metamodel.

- **Motivation:** This pattern provides a simple node mapping for the software systems. It is a combination of the `mapping` pattern and the `node refinement` pattern. It also discusses how one node in the source metamodel can be mapped to its corresponding element in the target metamodel.

- **Specification:**

  As illustrated in Figure 20, this transformation has four important parts. The `checkonly` part specifies that the `coreMM` metamodel is the source metamodel and its elements are read-only. This node of type `Node` can have one or more attributes (here, two of them are shown) that should be mapped into their corresponding elements in the target metamodel.

  The `enforce` indicates that `AOPLangMM` is the target metamodel and the node of type `Node` is transformed into two nodes in the target metamodel that are called `nodeT1` and `nodeT2`. It is left to the programmer to decide how one node in the source model can be mapped to how many nodes in the target model and how to map its attributes. This `enforce` part can also be written in another way as shown in Figure 19.

  In this case, the `node` of type `Node` in the `coreMM` is transformed into the `nodeT1`

```
transformation GenericAOPNodeMapping(coreMM:CoreMM, aOPLangMM:AOPLangMM){

top relation NodeMapping {
  attr1: AttributeType;
  attr2:AttribyeType;

  checkonly domain coreMM  node:Node{nodeAttr1 = attr1, nodeAttr2=attr2 };

  enforce domain aOPLangMM nodeT1:NodeTarget1{nodeT1Attr1 = attr1};

  enforce domain aOPLangMM nodeT2:NodeTarget2{nodeT2Attr2 = attr2};

  when {
        RootMapping(node.parent, nodeT1.parent);
        RootMapping(node.parent, nodeT2.parent);
  }

  where {
        ElementMapping(node, nodeT1);
        ElementMapping(node, nodeT2);
    }
}}
```

Figure 20: The Generic AOP node mapping pattern.

of type `NodeTarget1` in the `AOPLangMM`. The `nodeT1` also has two attributes, `nodeT1Attr1` and `nodeT2Attr2` that are created and are similar to `attr1` and `attr2` respectively.

The `when` clause checks the precondition. The node from one ecore can be mapped to another node in another ecore if their respective parents in node hierarchy are mapped to each other. This `when` clause checks and maps all the respective parents of the `node1` that are transformed into `nodeT1` and `nodeT2`.

In the `where` part, two other relations are invoked to map other elements in the source metamodel to the target metamodel.

7.3.2 CoreAOP model - AOP mapping pattern

The second pattern is dedicated to explain how to transform the crosscutting concerns that are modeled using CoreAOP to the crosscutting concerns in another AOP model that can be either an AOP language-specific model or the language-independent one. By adding the relations that are used to map the four important concepts in AOP (aspect, advice, pointcut and join points), this pattern is more specific than the previous one and it can be used to map the CoreAOP model to any AOP language-specific model at the lower level of abstraction or even any language-independent model at the same level of abstraction.

- **Goal:** To obtain the AOP language-specific or language-dependent model from the CoreAOP model.

- **Motivation:** This pattern provides a general solution for writing QVT transformation to transform a high level model that is modeled using the CoreAOP profile to a more detailed and specific model that is based on a specific aspect-oriented programming language. It also can be used to map a language-independent model to another model at the same level of abstraction.

- **Specification:** As illustrated in Figure 21, this pattern has four mapping relations for transforming aspect, advice, pointcut and join point to their corresponding constructs in another aspect-oriented language metamodel. The first relation is dedicated to an aspect and transforms an aspect in CoreAOP to aspect that is modeled using another AOP metamodel. The postcondition for

this relation is used to transform advice, pointcuts and join point to their corresponding constructs. We assume that in all AOP languages an aspect contains advice, pointcuts and join points.

The second relation is for transforming advice. For writing these transformations, the names referring to the elements in the `checkonly` domain part and the `enforce` domain part (ex. `coreAdvices` in `checkonly` and `advices` in `enforce`) have to be similar to the name used in the metamodel. In other words, in CoreAOP metamodel, the aspect contains a list of advice that is called `coreAdvices` and there is a list of advice in an aspect-oriented language dependent metamodel that is called `advices`.

The third and forth relations are for creating pointcuts and join points respectively. By providing a UML profile for an AOP language such as AspectJ or AspectC++, it is also possible to use the *CoreToAOPModel* pattern to provide the transformation for producing a language-specific model from CoreAOP model. Additionally, using this pattern, the reversible pattern (PSM to PIM) can be provided for reverse engineering of the system.

In the next Section, we illustrate how we use these patterns to specify model transformations for manipulating model of crosscutting concerns. In this thesis, the **Core Model - AOP Model** is used for writing the following model transformations for producing language-specific models from CoreAOP:

```
transformation CoreModel2AOPModel(coreMM:CoreMM, aOPMM:AOPMM) {

top relation CoreAspectToAOPAspect {
  pn: String;
  checkonly domain coreMM  a:Aspect{name = pn};
  enforce domain aOPMM  aj:Aspect{name = pn};

  where {
    CoreAdvice2AOPAdvice(a, aj);
    CorePoincut2AOPPointcut (a, aj);
        CoreJoinpoint2AOPJoinpoint(a, aj);
  }
}

// map coreAdvice to advice in another AOP model.
relation CoreAdvice2AOPAdvice  {
  an, cn, typeC, typeJ:String;

  checkonly domain coreMM a:Aspect {
    coreAdvices = adv:CoreAdvice{ name = an, adviceType = typeC}
  };
  enforce domain aOPMM aj:Aspect {
    advices = advJ:Advice{name = cn, adviceExecution = typeJ}
  };

  where {
   cn = an;
   typeJ = typeC;
   CorePoincut2AOPPointcut (a, aj, adv, advJ);
   CoreJoinpoint2AOPJoinpoint(a, aj, adv, advJ);
  }
}

//map core pointcut to AOP language pointcut.
relation CorePoincut2AOPPointcut  {
  an, cn :String;
  isAbs : Boolean;

  checkonly domain coreMM adv:CoreAdvice{
    pointcut = pcuts:CorePointcut{ name = an, isAbstract = isAbs}
  };

  enforce domain aOPMM advJ:Advice {
    perPointcut = pcutsJ:PointCut{ name = cn}
  };

  where {
    cn = an;
        CoreJoinpoint2AOPJoinpoint
          adv, advJ, pcuts, pcutsJ);
  }
}
```

Figure 21: The Core to AOP transformation pattern.

```
//map coreJoinpoints.
relation CoreJoinpoint2AOPJoinpoint  {
 an, cn, expr:String;
 isAbs : Boolean;

 checkonly domain coreMM pcuts:CorePointcut{
   joinpoints = jPoints:Joinpoint{ joinpointExpr = expr}
 };

 enforce domain aOPMM pcutsJ:Pointcut {
   perJoinpoints = jPointsJ:Joinpoint{ name = cn, operation = expr}
 };

 where {
   cn = an;
 }
}}
```

Figure 22: The Core to AOP transformation pattern - join point part.

- CoreAOP model to AspectJ: this transformation is proposed to transform CoreAOP models to AspectJ models.

- CoreAOP model to AspectC++: this transformation is proposed to transform CoreAOP models to AspectC++ models.

Figure 23 illustrates the relation between `Generic AOP node mapping pattern` and `Core Model - AOP Model` patterns and proposed model transformations.

7.4 Provide PIM to PSM model transformations

7.4.1 CoreAOP model to AspectJ

- **Goal:** To obtain AspectJ model from the CoreAOP model.

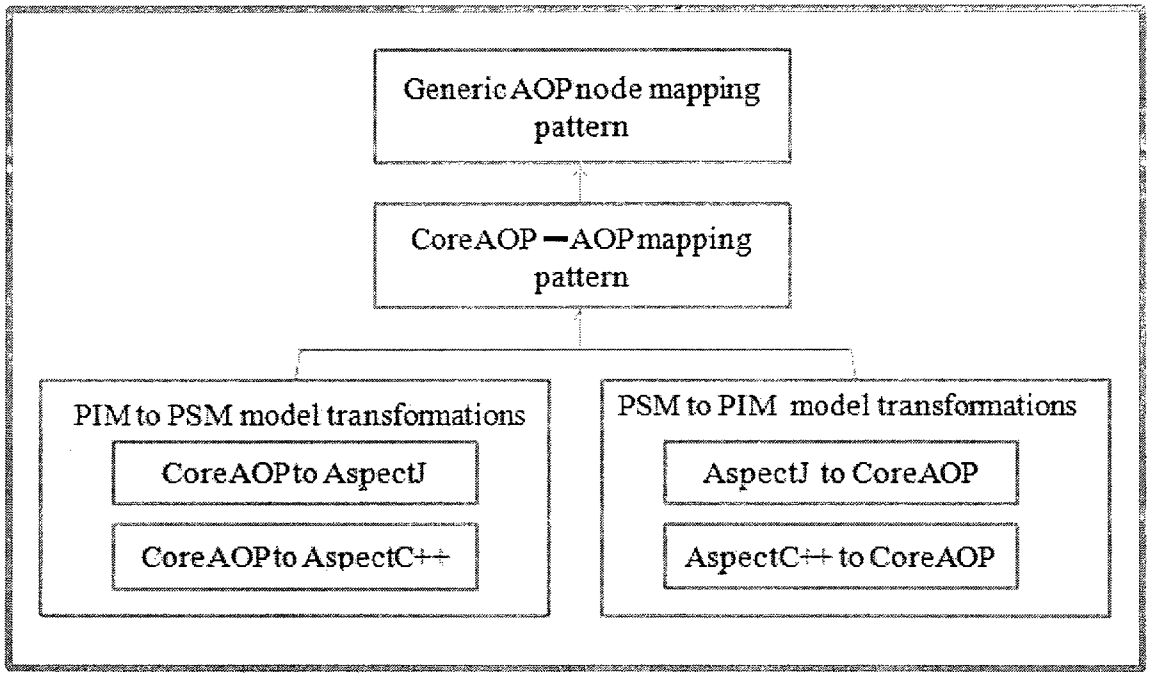

Figure 23: The relation between `Generic AOP node mapping pattern` and `Core Model - AOP Model` and the proposed AOP model transformations

- **Motivation:** This QVT transformation is used for transforming a high level model modeled using CoreAOP to an AspectJ model.

- **Specification:** In this transformation, the source model is the CoreAOP and the target model is an AspectJ metamodel disscussed in 7.2.1. The CoreAOP model to AspectJ mapping is shown in Figure 24. The first top relation is dedicated to create a package that is called `CrossCuttingConcern` in the AspectJ metamodel that we used. This package contains all aspects in this AspectJ profile. The top relation `CoreAspectToAspectJAspect` is for transforming the aspect in CoreAOP to the aspect in the AspectJ profile. This relation creates a new aspect that is called exactly the same as the aspect in the CoreAOP and invokes another relation for creating advice and pointcuts. The `CoreAdviceToAspectJAdvice` creates a new advice and sets its execution type that can be `after`, `before` or `around`. The `CorePoincutAndJoinpointToAspectJPointcut` relation is responsible for creating pointcuts and join points in AspectJ format. As illustrated in Figure 24, there are different types of pointcuts in AspectJ profile. It's developer's responsibility to select the specific type of pointcut to model the CoreAOP pointcut and its related join points.

7.4.2 CoreAOP model to AspectC++

- **Goal:** To obtain AspectC++ model from CoreAOP model.

- **Motivation:** This QVT transformation is used for transforming a high level

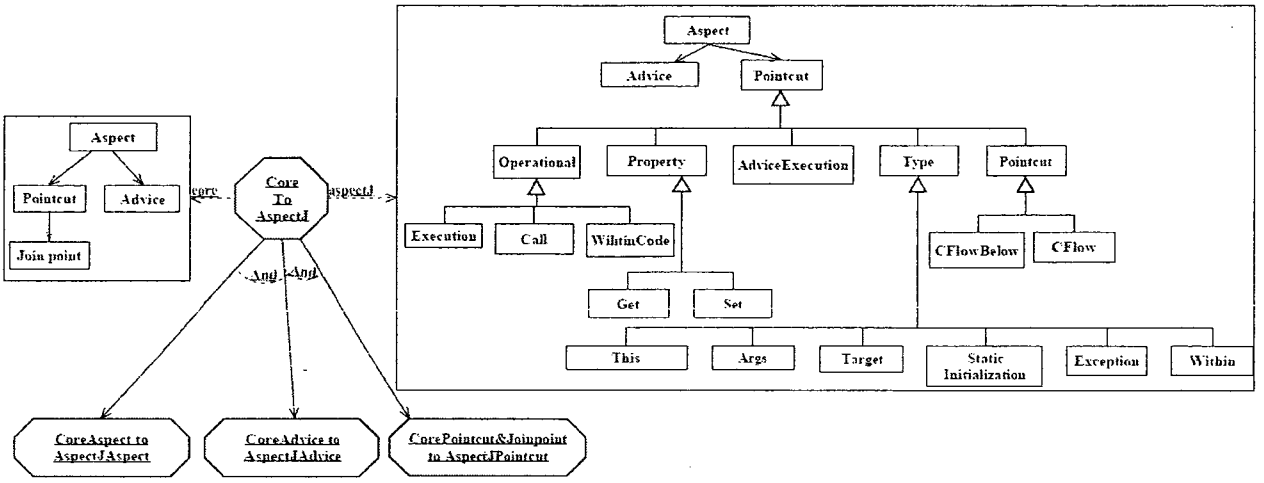

Figure 24: The Core to AspectJ mapping schema.

model modeled using the CoreAOP to the AspectC++ model.

- **Specification:** In this transformation, the source model is CoreAOP and the target model is an AspectC++ metamodel explained in Chapter 6. Figure 27 illustrates how CoreToAspectC++ transformation works. This transformation has four relations to map aspects, advice, joinpoints and pointcuts. The `CoreAspectToAspectC++Aspect` relation is similar to the AspectJ tranformation, therefore it is not shown here.

7.5 Provide PSM to PIM model transformations

Using the *CoreAOP model to AOP mapping* pattern, a transformation for reverse engineering the language-specific model to the language-independent model is provided.

```
transformation CoreToAspectJ
  (core:CoreAspect, aspectJ:AspectJCore) {

top relation CrosscuttingPackage {
  pn: String;
  checkonly domain core p:Package{name = pn};
  enforce domain aspectJ cc:CrosscuttingConcern {name = pn};
}

top relation CoreAspectToAspectJAspect {

  pn: String;
  checkonly domain core  a:Aspect {
          ownerPackage = p:Package {},
          name = pn};

  enforce domain aspectJ  aj:Aspect
  {ccPackage = cc:CrosscuttingConcern {},name = pn};

coreAdvices = adv:CoreAdvice {name = an, adviceType = typeC}
  where {
        ElementMapping(a, aj);
  }
}

relation ElementMapping {
  checkonly domain core  a:Aspect {};
  enforce domain aspectJ aj:Aspect {};

  when {CoreAspectToAspectJAspect(a, aj);}
        where {
          CorePoincut2AOPLangPointcut(a, aj);
          CoreAdvice2AOPLangAdvice(a, aj);
        }
  }
relation CoreAdvice2AOPLangAdvice  {
  an, cn, typeC, typeJ:String;

  checkonly domain core a:Aspect {
    coreAdvices = adv:CoreAdvice {name = an, adviceType = typeC}
  };

  enforce domain aspectJ aj:Aspect {
      advices = advJ:Advice {name = cn, adviceExecution = typeJ}
  };
```

Figure 25: The Core to AspectJ model transformation - part 1.

```
relation CorePoincut2AOPLangPointcut{
  an, cn, jExpr:String;
  isAbs : Boolean;

  checkonly domain core a:Aspect{
    pointcuts = pcuts:CorePointcut{ name = an, isAbstract = isAbs}
    joinpoints = jPoints:Joinpoint{ expr = jExpr}}
  };

  enforce domain aspectJ aj:Aspect {
    perPointcuts = pcutsJ:Pointcut{ name = cn}
    perPointcut = pcutsJ:CallPC{ name = cn}
    perPointcut = pcutsJ:WithinCodePC{ name = cn}

    perPointcut = pcutsJ:SetPC{ name = cn}
    perPointcut = pcutsJ:GetPC{ name = cn}

    perPointcut = pcutsJ:AdviceExecutionPC{ name = cn}

    perPointcut = pcutsJ:WithinPC{ name = cn}
    perPointcut = pcutsJ:ExceptionPC{ name = cn}
    perPointcut = pcutsJ:StaticInitPC{ name = cn}
    perPointcut = pcutsJ:TargetPC{ name = cn}
    perPointcut = pcutsJ:ArgsPC{ name = cn}

    perPointcut = pcutsJ:CFlowPC{ name = cn}
    perPointcut = pcutsJ:CFlowBelowPC{ name = cn}

    perPointcut = pcutsJ:CompositePC
              { name = cn, CompositionType = ''}
  }
  where{
    cn = an;
  }}
```

Figure 26: The Core to AspectJ model transformation - part 2.

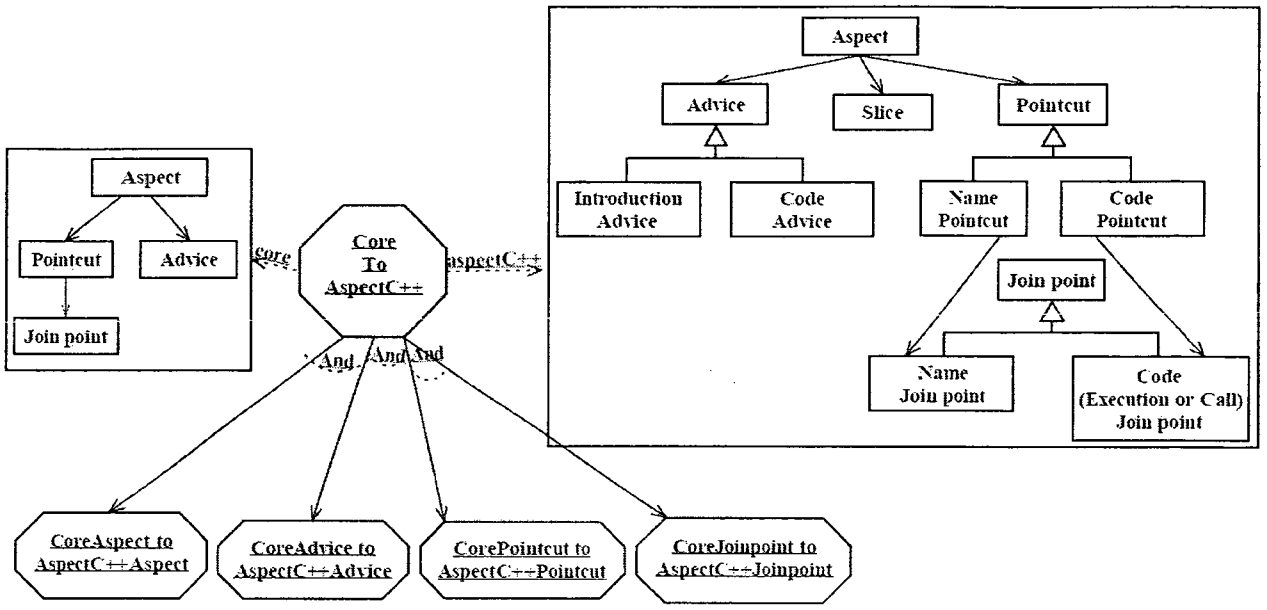

Figure 27: The Core to AspectC++ mapping schema.

In this transformation, the CoreAOP metamodel is the target and the language-specific model is the source, and some relations in the QVT language are specified to transform language-specific constructs to a generic model.

7.6 Discussion

7.6.1 The applicability of model transformations

The proposed transformations can be used to support the following activities during maintenance:

- **Forward engineering [6]:** transformation of a higher level specification into a lower level by transforming a set of model elements into a set of corresponding

```
transformation CoreToAspectC++(core:CoreAspect, aspectJ:AspectC++) {

relation CoreAdvice2AOPLangAdvice  {
  an, cn, typeC, typeJ:String;

  checkonly domain core a:Aspect {
   coreAdvices = adv:CoreAdvice{ name = an, adviceType = typeC}};

  enforce domain aspectC aj:Aspect {
   //codeAdvices = advJ:CodeAdvice { name = cn, adviceType = typeJ}}
   introAdvices =  advJ:IntroductionAdvice{ name = cn}};

   where {
     cn = an;
     //typeJ = typeC;
   }
}

relation CorePoincut2AOPLangPointcut {
  an, cn :String;
  isAbs : Boolean;

  checkonly domain core a:Aspect{
   pointcuts = pcuts:CorePointcut { name = an, isAbstract = isAbs}
  };
  enforce domain aspectC aj:Aspect {
   //namePointcuts = pcutsJ:NamePointcut { name = cn}
   codePointcuts = pcutsJ:CodePointcut
                     { name = cn, type = 'Execution or Call'}
  },

  where{
   cn = an;
   CoreJoinpointsToAspectCJoinpoints (pointcuts, codePointcuts);
  }
}
```

Figure 28: The Core to AspectC++ model transformation - part 1.

```
relation CoreJoinpointsToAspectCJoinpoints {
  an, cn, cExpr, cExpression:String;

  checkonly domain core pointcuts:CorePointcut{
    joinpoints = jps:Joinpoint { name = an, expr = cExpr}
  };

  enforce domain aspectC codePointcuts:CodePointcut {
    codeJoinpoints = codeJps:CodeJoinpoint
                  { name = cn, expression = cExpression}
  };

  where{
    cn = an;
        cExpr = cExpression;
  }
}
}
```

Figure 29: The Core to AspectC++ model transformation - part 2.

implementations is forward engineering. texttttCoreAOP model to AOP mapping pattern is a transformation pattern to support forward engineering and `CoreAOP model to AspectC++` and `CoreAOP model to AspectJ` are to exemplify how forward engineering is dont using the proposed transformations to produce a PSM model (for example, timingJ and debugJ) from a PIM model (timing and debug).

- **Reverse engineering [6]:** it is the inverse of forward engineering and it creates a representation of the system at a higher level of abstraction. Reverse engineering of a design model or code is done by abstracting away details related to specific programming languages and preserving the main concepts defined in the CoreAOP. The `CoreAOP model to AOP mapping` pattern also supports

reverse engineering by providing the reverse pattern. The coreAOP metamodel is used as the target model and the PIM model is produced after applying reverse transformation on PSM models that are produced using an AspectC++ or AspectJ metamodel.

- **Language migration [6]:** a transformation of a program written in one language into a program written in another language by preserving the level of abstraction is language migration. This transformation includes a reverse engineering for obtaining the PIM, and a forward engineering for adding constructs based on another programming language to provide the PSM model. To demonstrate the applicability of the proposed transformations in language migration, first we model a `Counter` aspect in AspectC++ that is shown in Figure 42. After that, we do reverse engineering and transform AspectC++ model to PIM model of the counter class and finally after doing forward engineering by applying `CoreAOP model to AspectJ` transformation, the AspectJ model of the `Counter` class is provided and displayed in Figure 43.

- **Reengineering [6]:** reengineering includes a reverse engineering for program comprehension and after applying changes and restructuring, a forward engineering is done for implementing new functionalities or modifying the system. All models used in this project, are either built automatically by applying model transformations or produced using proposed metamodels. Hence, all of these

models are in XMI format which enables current CASE tools to read, manipulate, apply changes and to visualize the instance models in UML. All activities involved in reengineering are supported by the proposed model transformations.

7.6.2 Preserve consistensy betweem models

Design by Contract (DbC) is a systematic approach in software engineering that proposed by [31]. DbC describes how elements of the software system collaborate with each other to satisfy the client (the user of the software system) and the supplier (the developer). In object-oriented programming, the client must provide a valid entry for the methods of the program. Therefore, satisfying the precondition of the methods is an obligation for client and a benefit for supplier.

Additionally, producing a valid result is an obligation for the supplier and a benefit for clients. In the specification of the model transformation in QVT, we have post conditions and preconditions. The `where` clause in the transformation specification checks the preconditions. Therefore, based on DbC, this is the obligation of the user of the transformation to select the valid input(source metamodel) for the transformations. In addition, the `when` clause in the transformation specification checks the post conditions. Hence, the developer who produced the transformation, is responsible for providing a correct result. If we have a valid metamodel as a source model for model transformations, the transformation produced the desirable models conforming to specific metamodel; these `where` and `when` clause guarantee the consistency between the source and the target model. If the transformation executed successfully, the

produced models are completely compatible with the source models and they provide another representations of the system at the same or different level of abstraction.

In addition, using QVT relations gives us automatic handling of traceability links [34]; therefore, it is possible to provide different models in different levels of abstraction automatically and preserve the consistency between them. When a rule/relation is executed, the transformation engine creates an internal structure that keeps the correspondence between the source and target elements. If we need to obtain the target element derived from a given source element, the QVT engine does this automatically [34]. Whenever a transformation is executed for the first time, these links are set, after that the transformation engine uses these links to check the consistency between different elements of the source and the target model.

Chapter 8

Case studies

To demonstrate the applicability of our proposal we have selected three case studies that are illustrated in this section. We follow the top-down approach and provide a conceptual model of these projects using the CoreAOP UML profile that can be used as an analysis model during the analysis and design phase of software development life cycle. Additionally, we demonstrate how the proposed model transformations can be applied to produce language-specific models.

8.1 Modeling crosscutting concerns

8.1.1 Graphical representation

It is possible to introduce new UML profile to the UML CASE tools and use that profile for modeling new concepts. We use MagicDraw and introduce the CoreAOP profile discussed in 11 as a new profile. Then, it is possible to use MagicDraw for

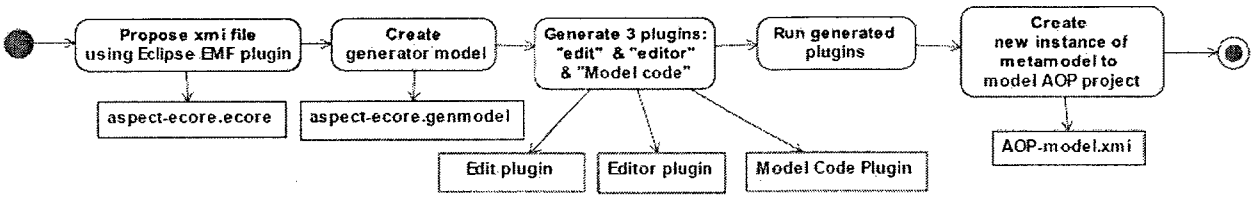

Figure 30: How to use the CoreAOP profile for modeling an AOP project in Eclipse EMF.

providing graphical representation for modeling aspects.

8.1.2 The XMI representation

The UML activity diagram shown in Figure 30 outlines the procedure by which the CoreAOP profile as an ecore file that is proposed by XMI format is used as a metamodel for modeling AOP constructs.

A genmodel file is created automatically from an ecore file using EMF which in turn produces three Eclipse plugins: `Edit`, `Editor` and `Model code`. The `Edit` plugin provides a flexible layer between the `Model code` and the `EMF editor` while the `Editor` plugin provides additional model-specific UI contributions to EMF. By running the generated plugins together, the proposed metamodel is added to the models that are provided by EMF. By opening a new EMF project, it is possible to create a new model of CoreAOP type and model crosscutting concerns.

After generating the plugins, a new model is added to the model wizard in Eclipse. After adding CoreAOP, the CoreAOP model wizard can now be used to create a new instance of the aspect model. The newly created CoreAOP model is opened in the main view in Eclipse and it is possible to add a new child that is a new aspect to this

model. Finally, it is possible to add advice and pointcuts to this aspect.

8.2 Applying model transformations

For writing and executing the QVT transformation, version 3.4.1 of the Eclipse SDK is used. Additionally, Eclipse Modeling Framework (EMF 2.4) and QVT 0.7 are used for working with transformations.

The procedure by which the QVT transformation is proposed in the Eclipse Modeling Framework is outlined as follows [29]:

1. Choose a Java project, a plug-in project or an EMF project.
2. Add *QVT Relations Nature* to the project.
3. Add Model registry to the porject. The model registry allows developers to specify a metamodel and give it a name. In this work, we have three model registries in XMI format:
   (a) *CoreAOP.ecore.*
   (b) *AspectJ.ecore.*
   (c) *AspectC++.ecore.*
4. Create new ".qvtr" file and write the transformation.
5. Launch the project in Eclipse.
6. Create new specification:

(a) Select the wanted transformation.

(b) Select the source model. For PIM to PSM transformation, the source model is an aspect modeled with CoreAOP profile, such as `timing.xmi`.

(c) Select the target model.

7. Run the transformation.

8.3 Case study 1: Telecom

We deploy our approach over a small-scale project that is provided in the Eclipse AspectJ Development Tools project [1]. Telecom simulates a telecommunication system and it contains 731 lines of code. The following classes are provided to implement the Telecommunication system.

- **Customer:** It is a caller or receiver of the call specified by its name and the area code. It also has protocols for managing calls like call, pickup and hang up.

- **Connection:** It is a circuit between `customers` that can be either long or local distance. This connection is held between two customers.

- **Call:** A call supports the process of a customer trying to connect to others. It is held between two `customers` by creating a `connection` between them. One call can contain one or more `connections`.

- **Abstract Simulation:** It is responsible for executing the telecommunication system, connecting different customers. Moreover, it provides complete reports of customers' activities. This class has 3 subclasses to create objects and puts them to work.

    1. **BasicSimulation:** It implements the `AbstractSimulation.run(..)` method to simulate the execution of the program.
    2. **BillingSimulation:** This subclass implements the `AbstractSimulation.report(..)` method to print a report that contains the connection time and the bill for a customer.
    3. **TimingSimulation:** This subclass implements the `AbstractSimulation.report(..)` method to print a report of the connection time.
    4. **Timer:** Timer class simulates a simple timer machine for calculating the elapsed time of each call.

The Telecom UML graphical representation that is produced by MagicDraw is shown in Figure 31. To log the activities and to provide appropriate billing for each customer, three different aspects are presented:

- `Timing` aspect calculates the duration of a connection and the total time for each call per customer.
- `TimerLog` aspect provides a complete report of `Timer` class's activities.

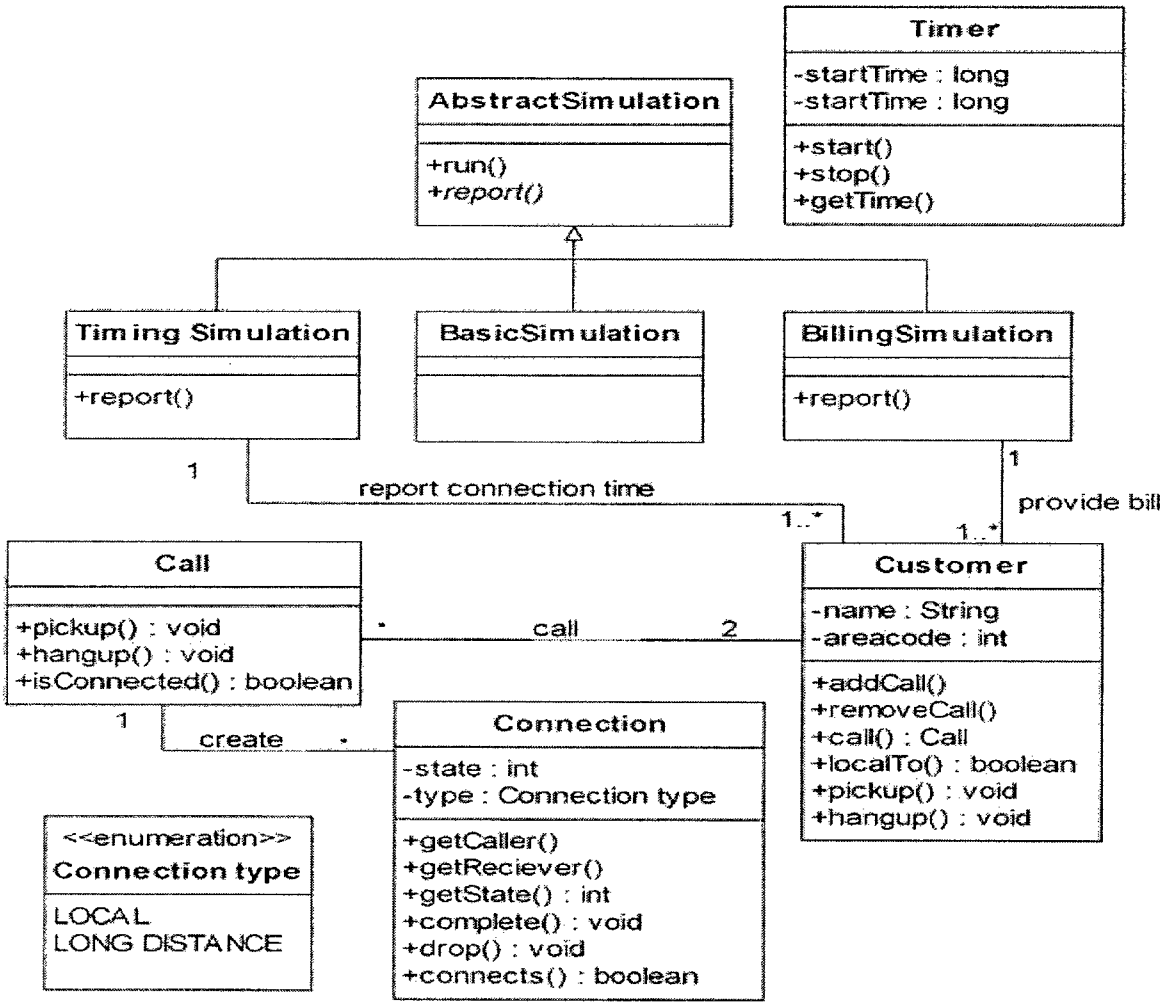

Figure 31: The Telecom case study class diagram.

- `Billing` aspect is responsible for providing a complete call report that includes call details, duration and expenses for customers based on the timing and the type of connection.

8.3.1 Provide a PIM model of the Telecom system

To demonstrate the use of our profile, we show in this case study how it can represent the three Telecom aspects. As modeled in Figure 32, there are three classes which are stereotyped as aspects. If we look at the `Timing` aspect, it has two operations that are abstract and stereotyped as pointcuts. The name of the corresponding join points for these pointcuts are shown as operation arguments as illustrated in Figure 14. This aspect has two methods that are stereotyped as advice and their names are identical to their pointcuts' name (`complete` and `endTiming`). Additionally, the type of each advice is shown as a method property.

As the collaboration between an aspect and other components of the system is performed through join points, we attach a UML interaction diagram to each aspect's definition (See Figure 14) to illustrate this collaboration. In the timing aspect, after a call to the `drop()` method of `Connection` class, the `endTiming` advice executes. Furthermore, after a call to method `complete()` of class `Connection`, the `complete` advice executes.

Additionally, using Eclipse EMF, a XMI representation for each aspect in Telecom is provided using the CoreAOP profile. As an example, the XMI representation of the `timing` aspect is illustrated in 33. Eclipse EMF plugin illustrates XMI file in a

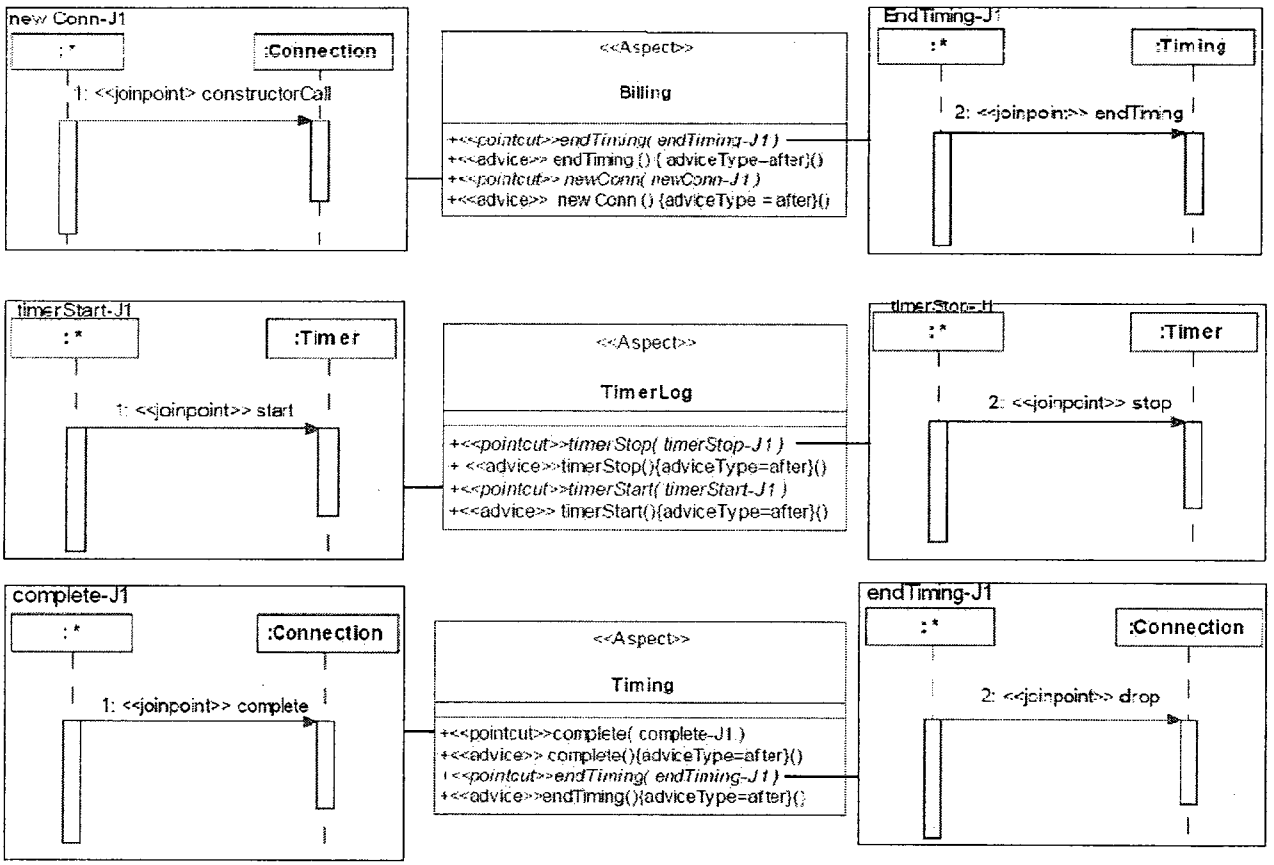

Figure 32: Modeling the Telecom case study.

tree-based format.

8.3.2 Apply a model transformation to provide PSM models

We apply the *CoreAOP model to AspectJ* transformation into the Telecom language-independent model. As we discussed in 8.2, for this transformation we use two model registries.

1. `CoreAOP.ecore` as a source metamodel. This metamodel is used to provide PIM model of Telecom aspects such as Timing 33.

2. `AspectJ.ecore` as a target model.

For this transformation, the PIM models of Telecom such as `timing.xmi` 33 are the source models and we produce an AspectJ models after executing the transformation.

After executing the *CoreAOP model to AspectJ* transformation, the PSM model of Timing aspect is built. As illustrated in Figure 34, the language-independent model of the Timing aspect contains 2 advice:

1. The `complete` advice in `timing.xmi` is transformed into the `complete` advice in AspectJ model of Timing aspect.

2. The `endTiming` advice in `timing.xmi` is transformed into the `endTiming` advice in AspectJ model of Timing aspect.

Additionally, this transformation map two pointcuts in PIM model of Timing to the two `composite` pointcuts in AspectJ:

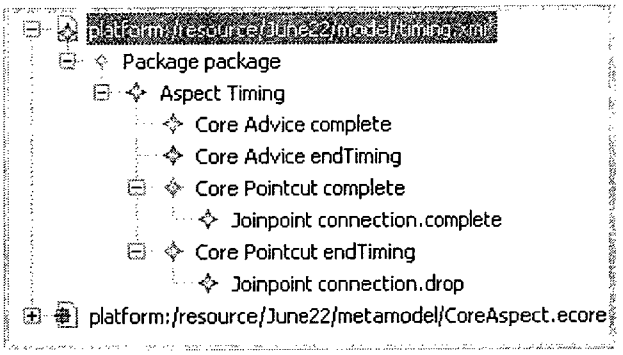

Figure 33: The language-independent model of Timing aspect.

1. The core pointcut `complete` is transformed into the `composite pointcut complete` in AspectJ that has two parts:

   (a) `Call pointcut` that is modeled calling the method `complete` of the `Connection` class.

   (b) `target pointcut` to specify that this pointcut needs the reference to the object of `Connection` class.

2. The core pointcut `endTiming` is transformed into the `composite pointcut endTiming` in AspectJ that has two parts:

   (a) `Call pointcut` that modeled calling the method `drop` of the `Connection` class.

   (b) `target pointcut` to specify that this pointcut needs the reference to the object of `Connection` class.

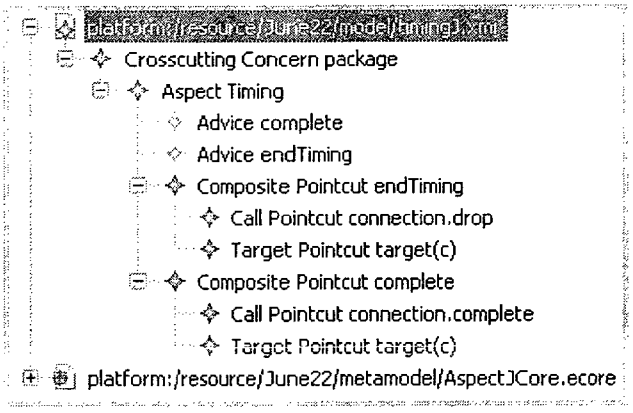

Figure 34: The AspectJ model of Timing aspect after executing transformation.

8.4 Case study 2: Spacewar

Spacewar is a medium-scale AOP project provided by the Eclipse AspectJ Development Tools project [1] and has been deployed in the literature as a benchmark [5, 44]. At 2300 lines of code, Spacewar simulates an arcade asteroids game and it shows a variety of interesting uses of aspects. In this game, a spaceship represented by a movable triangle is controlled by a user and tries to eliminate the other spaceships. The other spaceships have the same triangle form in different colors. Spacewar contains two packages:

1. `coordination` contains five classes, three interfaces and one aspect, namely `Coordinator`.

2. `spacewar` contains ten classes and five aspects. Additionally, there are three aspects that can be considered as AspectJ compilation units as the extension of their files are ".aj".

By instantiating the `Game` class or calling the `main` method, the spacewar game starts. `SpaceObject` is an abstract class to simulate objects that float around in space. It has information about the position, the velocity and the size of the objects. The `SpaceObject` adds itself to the registry after it is constructed and when it dies, a `SpaceObject` removes itself from the `Registry`. When class `Game` is created, the subtypes of this class (`Ship`, `Bullet` and `EnergyPacket`) are created.

8.4.1 Provide a PIM model of the Spacewar game

These aspects are defined in this project to ensure synchronized access to critical methods of the game in the presence of several threads.

- The `Coordinator` is an abstract aspect that provides the basic functionality for synchronizing and coordinating different threads upon entering and exiting methods [1]. By marking critical section methods in an object as self-exclusive or mutually exclusive, the threads will be synchronized. It has two subclasses that implement it: `RegistrySynchronication` and `GameSynchronization`. Figure 36 illustrates the modeling of the `Coordinator` aspect with its subclasses.
  - The `RegistrySynchronization` aspect guarantees synchronized access to methods of the `Registry` while threads are running. For each instance of the `Registry` class there is one instance of this class.

  - The `GameSynchronization` aspect guarantees synchronized access to methods of the `Game` while threads are running. For each instance of the `Game` class there is one instance of this class.

- The `RegistrationProtection` is a static inner aspect which is defined inside `Registry` class in `Registry.aj` file. It is responsible for keeping track of all the `SpaceObjects` that are floating around by supporting the following operations: register and unregister.

- The `Display` aspect draws the space object on the screen and indicates how much space it occupies. The display provides the look of the Game by displaying the game as it goes along. This aspect is illustrated in Figure 35. This aspect contains 5 unnamed pointcuts that are shown as an abstract operation stereotyped as pointcut. Join points are illustrated in sequence diagrams and define the specific events in the execution of the program. For example, whenever an instance of the `Display`, `Game` and `Player` are created the corresponding advice is triggered.

- `Debug` is used for debugging the Spacewar project and displays the tracing information to the output. The developer can enable or disable this aspect by weaving or not weaving it. The model of this aspect is depicted in Figureẽffig:debug.

- `EnsureShipIsAlive` checks whether the ship is alive before performing any console commands.

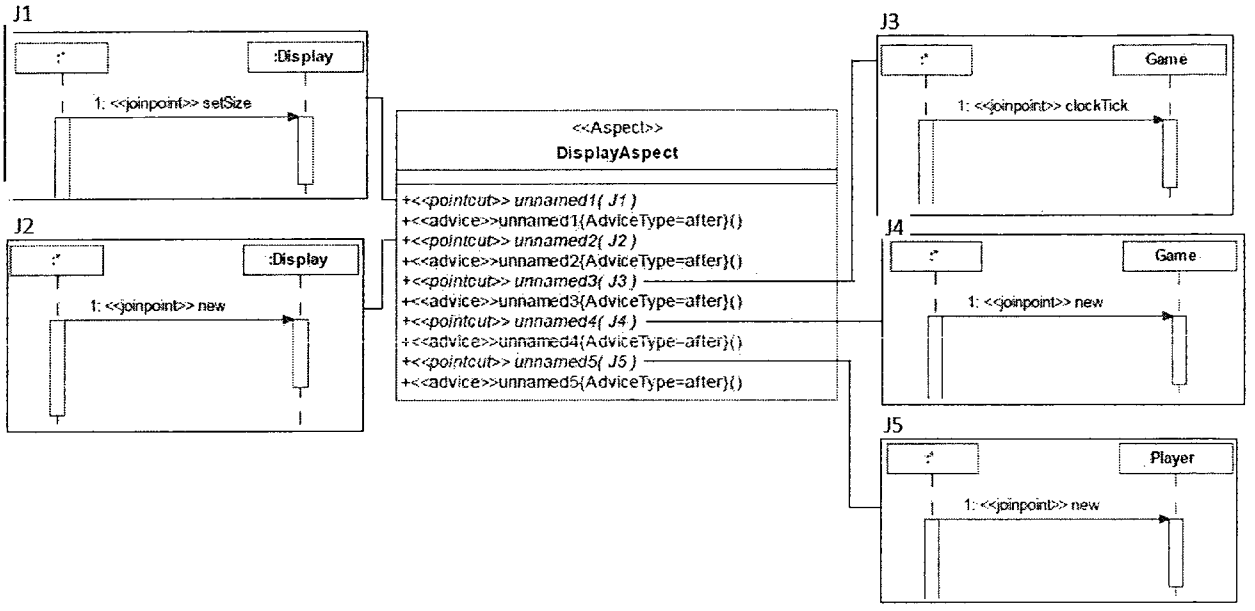

Figure 35: Modeling the Spacewar case study - Display aspect.

- `SpaceObjectPainting` is a static inner aspect which is defined inside `Display1` class in `Display1.aj` file. It sets different colors to different objects such as `Ship`, `Bullet` and `Energy packets` to display them.

The SpaceObjectPainting and the EndureShipIsAlive models are shown in Figures 38. In this case study, the aspects are modeled to show how the profile can be applied to a large-scale AOP project.

All aspects in this project are modeled completely using CoreAOP profile that is introduced as a new profile to MagicDraw. In aspects with no pointcut expressions, such as `RegisterationProtection`, the join points are directly attached to the advice.

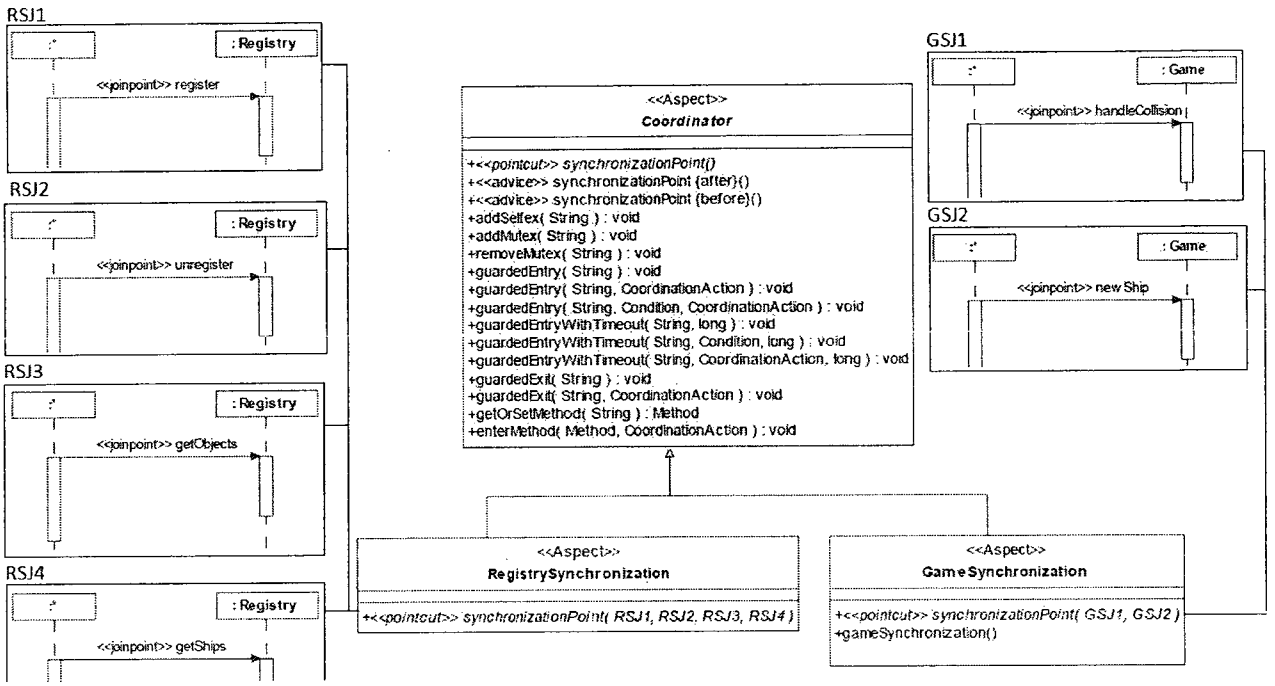

Figure 36: Modeling the Spacewar case study - Coordinator aspect.

Additionally, using Eclipse EMF, a XMI representation of each aspect in Spacewar is provided using the CoreAOP profile. As an example, the XMI representation of the debug aspect is illustrated in 39.

8.4.2 Apply a model transformation to provide PSM models

We apply the *CoreAOP model to AspectJ* transformation into the Spacewar language-independent model. As we discussed in 8.2, for this transformation we use two model registries.

1. `CoreAOP.ecore` as a source metamodel. This metamodel is used to provide PIM model of Spacewar aspects such as Debug 39.

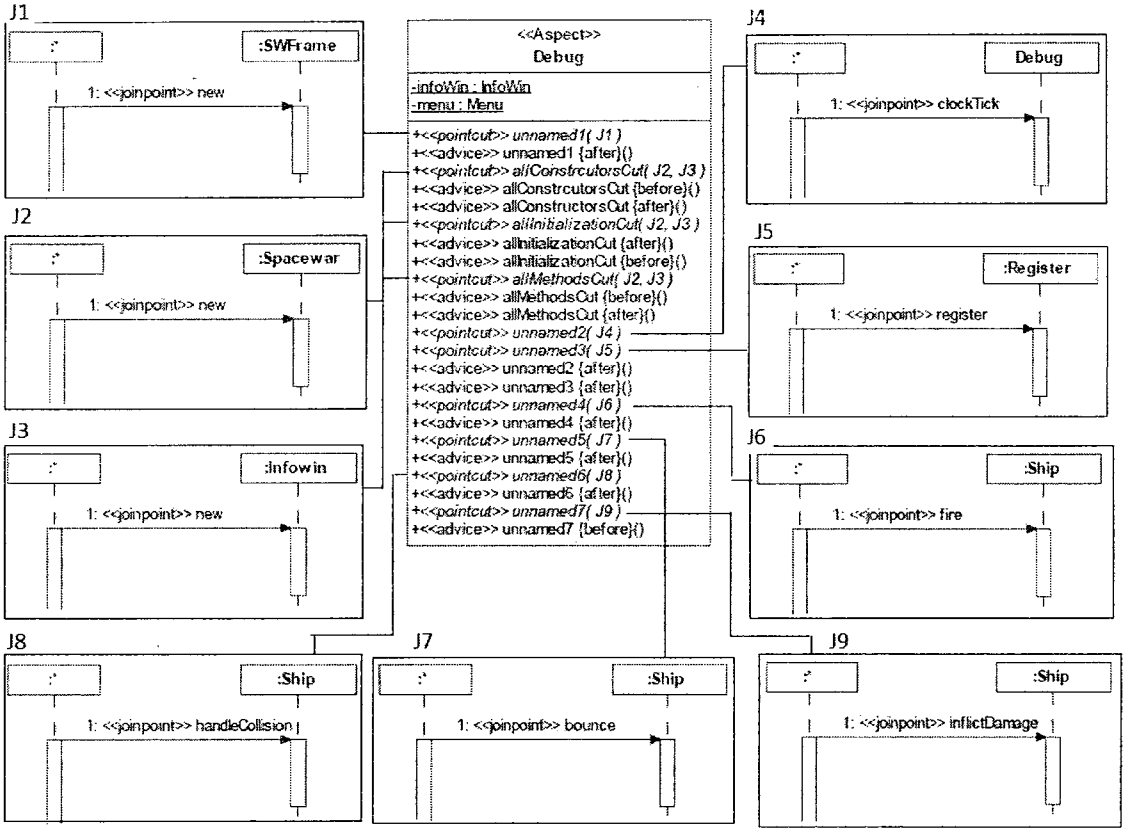

Figure 37: Modeling the Spacewar case study - Debug aspect.

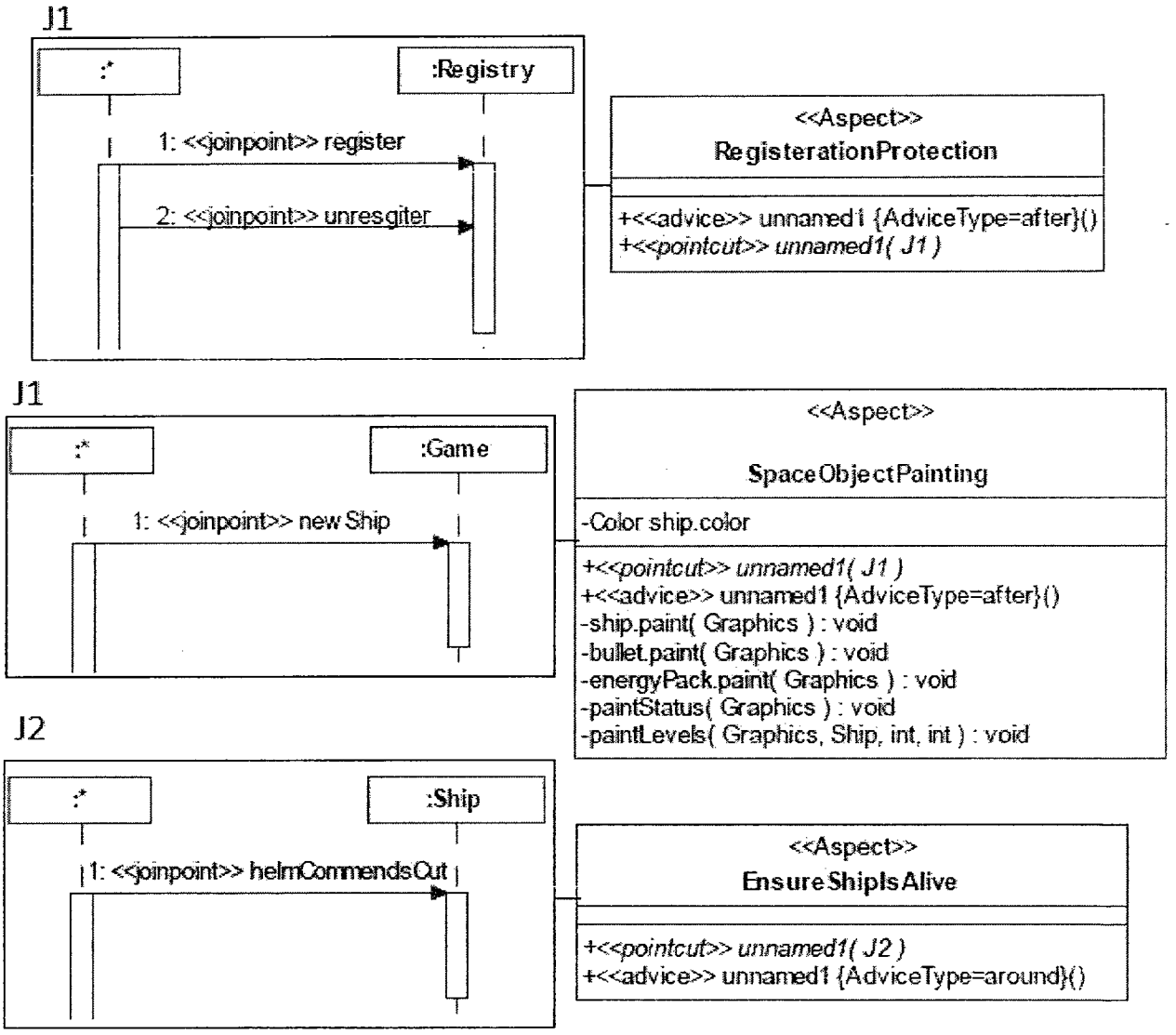

Figure 38: Modeling the Spacewar case study - RegisterationProtection, SpaceObjectPainting, EndureShipIsAlive aspects.

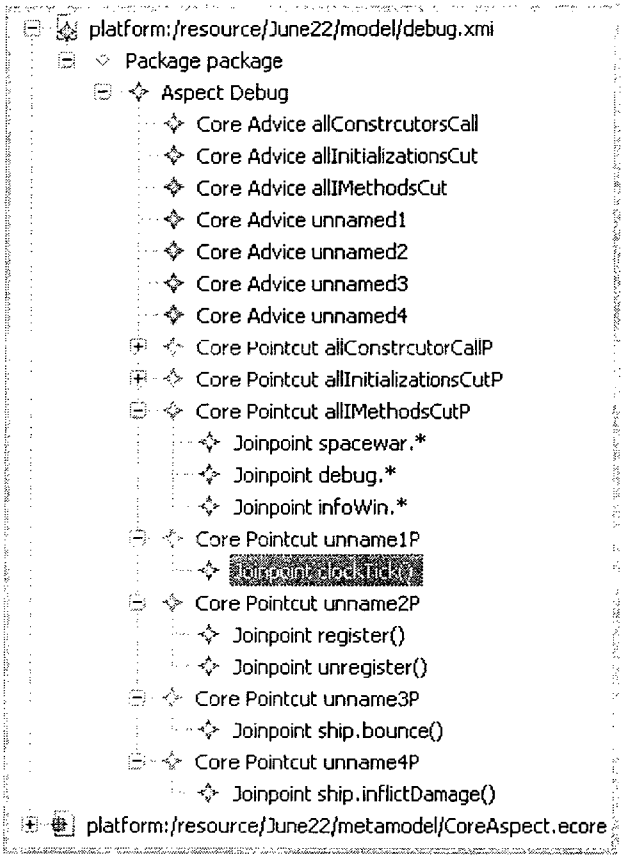

Figure 39: The PIM model of Debug aspect.

2. `AspectJ.ecore` as a target model.

For this transformation, the PIM models of Spacewar such as `debug.xmi` 39 are the source models and we produce an AspectJ models after executing the *CoreAOP model to AspectJ* transformation. A language-independent model of *Debug* aspect is transformed into AspectJ model automatically using proposed QVT transformation. The PIM model and the PSM (AspectJ) model are displayed in Figure 39 and Figure 40 respectively.

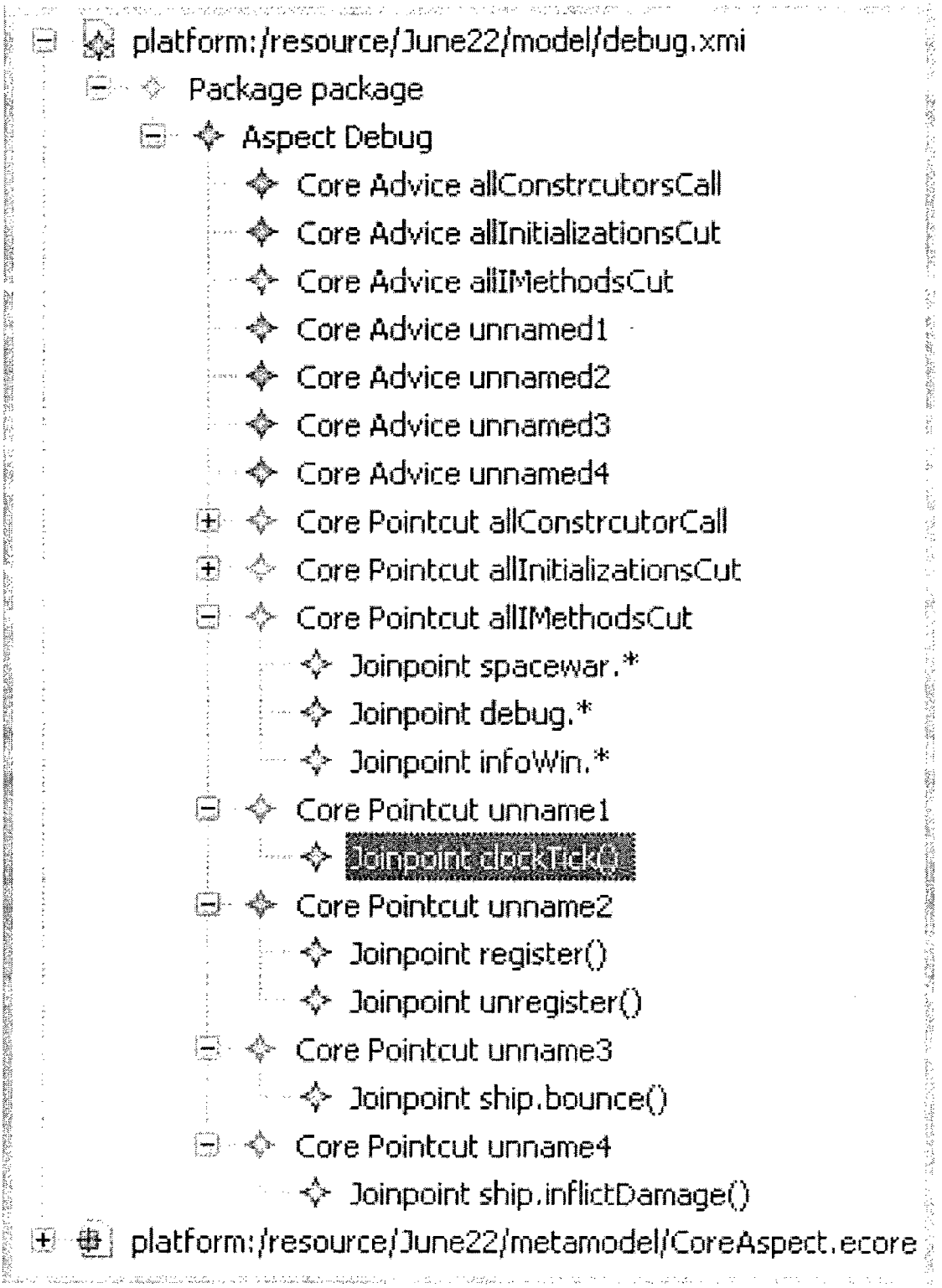

Figure 40: The PSM model of Debug aspect after executing transformation.

Whenever there are more than one pointcuts in one aspect, there are two ways for writing the model transformation:

1. These pointcuts have the same kind of join points: The programmer selects a set of pointcuts to model all pointcuts toghether. For example, for transforming the PIM model of timing aspect to AspectJ model, the programmer only selects the two types of pointcuts (`CallPC` and `TargetPC`) to execute the transformation.

2. These pointcuts have different kind of join points: It is necessary to model each pointcut separately, and after that combine all PSM model together such as `debug` aspect. We suggest that in this case, the programmer modeled all pointcuts as `Composite` pointcuts and then modify the model using the EMF editor to add different types of pointcuts to the `Composite` pointcut. In this way, all aspects with their advices are executed automatically and it is only necessary to modify the pointcuts manually using EMF editor.

8.5 Case study 3: Counter aspect in AspectC++

The `Counter` aspect is an aspect for calculating the number of object instantiations in a simple C++ program. In this aspect, whenever an object of `Polygon` or `Circle` classes is instantiated, the counter increases by one. When the program terminates, because of applying the advice code for method `main`, the final number of object instantiation is displayed to the programmer.

8.5.1 Provide a PIM model of the Counter aspect

We provide a PIM model of a Counter aspect using CoreAOP profile in XMI format, this model is illustrated in Figure 41. In this aspect, we have 2 advice and 2 pointcuts that describe as follows:

1. The first advice is called `unnamed`.
2. The second advice is called `counted`.
3. The first pointcut is called `unnamed` and whenever this pointcut is reached the `unnamed` advice is triggered. This pointcut contains one joinpoint that monitors the execution of the `main` function.
4. The second pointcut is called `counted` and whenever this pointcut is reached the `counted` advice is triggered. This pointcut has two joinpoints that are responsible for monitoring object instantiation of the `Circle` and the `Polygon` classes respectively.

8.5.2 Apply a model transformation to provide PSM models

We execute *CoreAOP model to AspectC++* transformation to do forward engineering. For this transformation, we use two model registries.

1. `CoreAOP.ecore` as a source metamodel. This metamodel is used to provide PIM model of Counter aspects.
2. `AspectC++.ecore` as a target model.

After executing the *CoreAOP model to AspectC++* transformation, the PIM model of the `counter` aspect (see Figure 41) is transformed to the AspectC++ model that is illustrated in Figure 42. This transformation maps the `unnamed` advice to the `Code advice` in AspectC++ that is also called `unnamed`. Because this advice is triggered after the exection of the `main` function, based on the definition of the `Code Advice` in AspectC++, this is modeled as a code advice. This `Name advice` has two `Name joinpoints` that are worked on the `Circle` and the `Polygon` classes.

The `counted` advice in PIM model is modeled as an `intorduction` advice in PSM model.

To do reverse engineering of this aspect, we apply *CoreAOP model to AspectC++* in the reverse direction. It is only necessary to change the target and the source metamodels. After applying the reverse engineering, the PIM model of a Counter aspect is produced.

Finally, we execute *CoreAOP model to AspectJ* transformation to do forward engineering for providing PSM model of Counter aspect for AspectJ. For this transformation we use two model registries.

1. `CoreAOP.ecore` as a source metamodel. This metamodel is used to provide PIM model of Counter aspects.

2. `AspectJ.ecore` as a target model.

As illustrated in Figure 43, the AspectJ model of the Counter aspect has two advice and two `composite` pointcuts. The pointcut that is called `unnamed` has two

`Call pointcuts` to capture any constructor's call to Circle and Polygon classes. The second pointcut is called `counted` that contains one `execution` pointcut that captures execution of the main.

This case study illustrates that how it is possible to do forward engineering and reverse engineering of an aspect using the proposed model transformations. Additionally, in this example, we display the language migration; we map the AspectC++ model of the Counter aspect to the AspectJ model using an intermediate PIM model. First, we provide a PIM model of the Counter aspect using CoreAOP. After that, by applying *CoreAOP model to AspectJ* model transformation, the PSM model of the Counter aspect is provided. We use *AspectJ model to CoreAOP* as a reverse transformation to provide a PIM model of Counter aspect and Finally, we apply *CoreAOP model to AspectC++* to the PIM model to produce AspectC++ model of the Counter aspect.

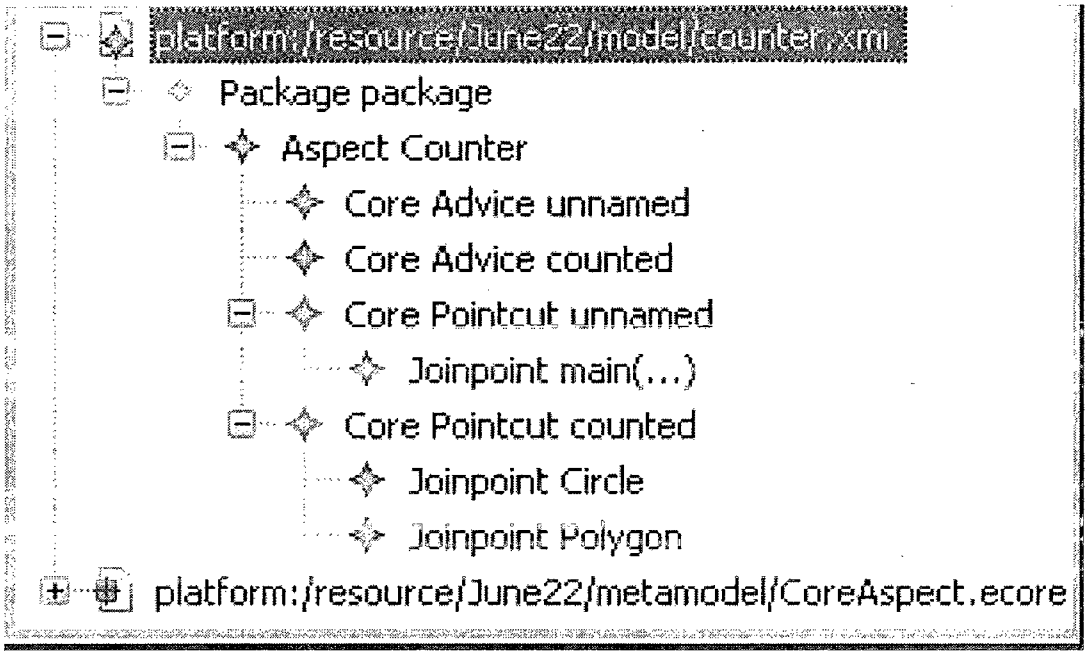

Figure 41: The PIM model of Counter aspect.

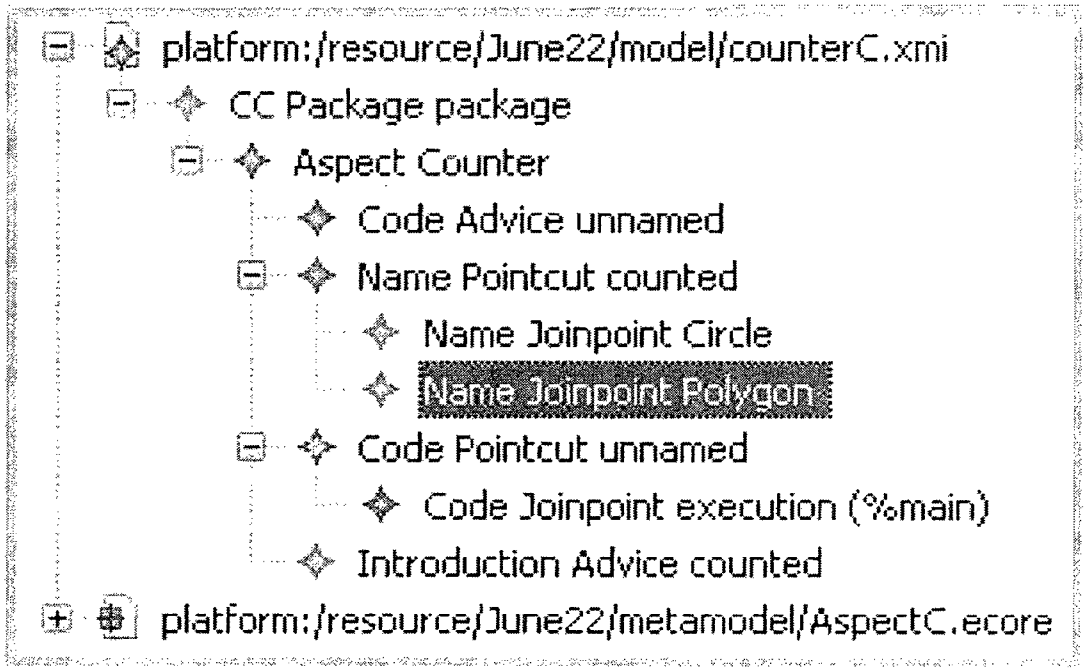

Figure 42: The AspectC++ model of Counter aspect after executing transformation.

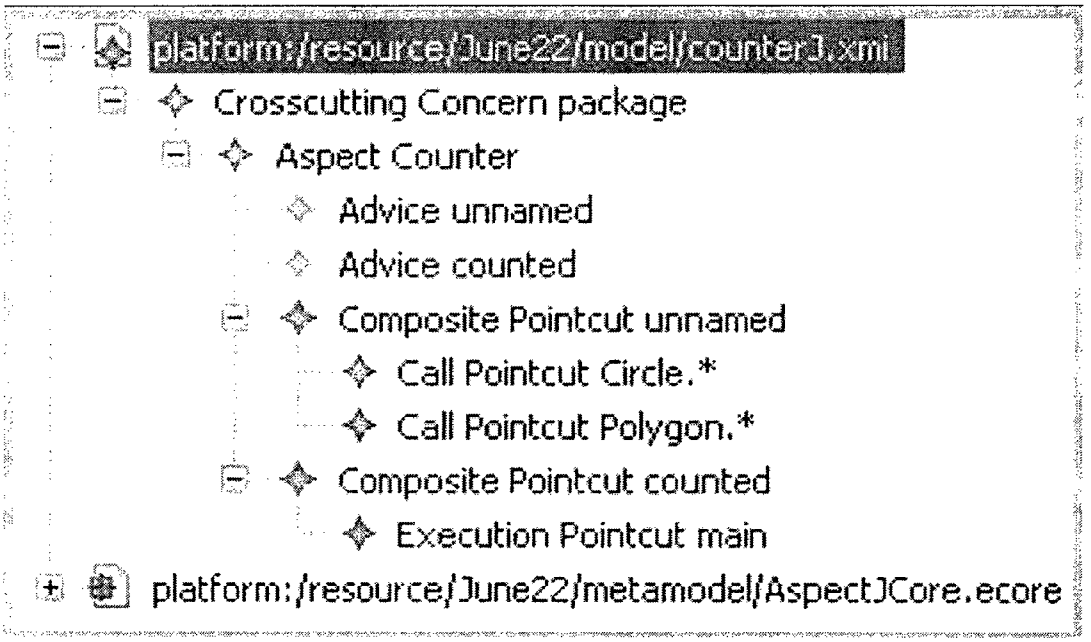

Figure 43: The AspectJ model of Counter aspect after executing transformation.

Chapter 9

Related work

The approaches that are relevant to our work can be categorized into two main groups. The fisrt group focuses on modeling aspects, whereas the second group explores how to provide transformation patterns and model transformations. These two groups are presented in Section 9.1 and Section 9.2.

9.1 Modeling crosscutting concerns

In the last few years, UML profiles have been proposed for modeling crosscutting concerns. Aldawud *et al.* [4] argue how UML profiling would be a viable approach for modeling crosscutting concerns. Reina *et al.* [41] provide a survey of some other works. These were found to be either language-dependent or provide no support for aspectual behavior. Other works such as [12, 39, 19, 51] are based on the AspectJ programming language as a popular general-purpose aspect-oriented programming

language.

Albunni *et al.* [3] propose the use of a UML activity diagram for modeling aspects in web applications. However, they do not provide a UML profile for modeling crosscutting concerns. Zhou *et al.* [56] model dynamic behavior of crosscutting concerns using sequence diagrams. In contrast with our approach, they do not provide a model for static behavior of crosscutting concerns. Since the dynamic behavior of crosscutting concerns affects the execution of core components, they modify existing sequence diagrams by introducing additional crosscutting bars.

A platform-independent behavioral model is proposed in [32]. In this model, advice is modeled as a stereotyped operation. Additionally, the authors follow a rather complex expression-based approach for modeling join points and it contains specific details about join points that are based on different aspect-oriented technologies. Fuentes *et al.* [15] model advice as a common procedure using a UML activity diagram without an input object. In this model, an advice is placed in an aspect definition as a stereotyped method. The authors provide actions for the advice that are used for retrieving data related to the join point.

Coelho *et al.* [3] suggest using software visualizations approach to model aspects. They present dynamic aspect diagrams to display the influence of an aspect on an existing software system. This approach is completely different from using UML extension to model crosscutting concerns by providing dynamic aspect diagrams. There is no tool support for this approach and it is not shown to be expressive or general enough to support specific modeling.

Our work is partially similar to [15, 19] though it differs conceptually regarding the modeling of crosscutting concerns. Advice cannot be considered as a method or operation, hence our proposed profile models advice as a stereotyped behavioral feature. Furthermore, in order to produce an executable model, they ([15, 19]) propose a weaving procedure on the model itself for combining crosscutting concerns and non-crosscutting concerns.

The idea to define a profile to support crosscutting concerns modeling (CoreAOP profile 5.2) in UML was inspired by [12] that is an AspectJ profile. However, for proposing the CoreAOP, we eliminat all constructs specific to the AspectJ programming language and support only the most essential constructs that make a system AOP. Our approach enables more kinds of reengineering and presumably makes the profile easier to explain, understand and use. Also, the CoreAOP profile proposed in this thesis can easily be extended to support any additional features specific to a particular aspect-oriented language. What's more, in contrast with other proposed profiles, we dedicate a specific icon for displaying crosscutting concerns.

9.2 Model transformation

An approach to weave Java classes and AspectJ aspects at the modeling level is proposed in [2]. They propose a set of transformations to weave AspectJ models and the main functionality of the software system to produce a Java model before the code generation phase to deal with non aspect-oriented platforms.

Tekinerdogan *et al.* [52] provide systematic analysis to display the impacts of separation of concerns in MDA. Using a specific case study, they define an abstract model of transformation by proposing a set of transformation patterns. These transformations explain how model A can be transformed into model B while different concerns are included in these models. They analyse the impact of applying model transformation for mapping different models to each other (PIM to PSM, PSM to code), in the specific case study, the concurrent versioning system (CVS). Finally, they propose some useful recommendations for coping with concern evolution in the MDA process.

Koehler *et al.* [24] investigate model-driven transformations using graph-based method, to map business view models into IT architectural models. This work like ours aims at supporting the complete development cycle and deals with different methods at the various levels of abstraction. However, the focus lies on service-oriented architectures with Web service that makes this approach specific for IT services.

A security aware Model-driven development based tool is proposed by [42], which allows them to produce the secure platform independent definition by applying secure policies.

Transformations proposed in previous works either are not executable [52] or are dependent on specific applications [24, 41, 48]. Solberg *et al.* [48] propose a conceptual model for model driven system and talk about how to transform the PIM model to the PSM model in a specific CORBA application. They work on the specific case-study

and they do not provide any reusable automatic transformation.

Transformation patterns that they propose are concerns transformations and they talk about how it is possible to compose each concern to model and build a composite model in a general way [52]. Also, there is no specific guideline for providing transformation models to work with crosscutting concerns.

Judson *et al.* [18] propose a pattern-based transformation declaratively at the metamodel level. They provide an extension to the UML metamodel to support model transformations. These works do not explicitly provide model transformation in an executable format and do not give details about transformation rules.

The idea to define transformation patterns using QVT to facilitate writing model to model transformation was inspired by Iacob *et al.* [17]. They identify basic transformation patterns and implement them using QVT; additionally they demonstrate how these transformations can be used for providing more complex model to model transformations. In our work, we focus on providing transformation patterns that can be used to map crosscutting concerns specifically.

9.3 Discussion

Approaches to support crosscutting concerns in MDA by means of model transformations are language-specific [2, 12, 19, 39, 51], or they are proposed for specific applications such as IT services or web applications [24], or the focus in these works lies on specific kinds of aspects such as security [43]. Our approach is high-level and

completely language-independent.

UML modeling was done using the Eclipse modeling framwork and it supports full XMI export/import capabilities. MagicDraw2 [1] is used to elaborate the graphical diagrams of this project. One limitation of our approach is a clear lack of effective and seamless tool support for our approach. We integrate different plugins to support modeling aspects and provide model transformations. Writing model transformations is considered to be semi-automatic; it is up to the programmer to change the proposed pattern and select language-dependent constructs to make sure that the program performs correct transformations.

[1] http://www.magicdraw.com/

Chapter 10

Conclusion and recommendations for further work

The primary motivation for this thesis arises from the fact that combining MDA and AOP increases maintainability of the system because of better separation of concerns. These two approaches have been proposed in order to improve software adaptability to changes. MDA enhances the adaptation to different technologies by means of three different levels of modeling while AOP improves modularization of crosscutting concerns at early stages of the software development process, which leads to more consistent, reusable and maintainable artifacts.

In this work, we explicitly address crosscutting concerns at different levels of software developement process at various levels of abstraction. We propose a language-independent UML profile for modeling both core and crosscutting concerns. The constructs in this profile are independent from any specific programming language

and thus can be used to support generic aspect-oriented modeling. Moreover, in this project a graphical notation schema is proposed to display crosscutting concerns. Being UML, it is already accessible to a wide users, yet still powerful enough to model crosscutting concerns precisely. Additionally, this proposal specifies crosscutting concerns without requiring any textual specification. This, in conjunction with the decision to use the XMI format, means that it is possible to manipulate, visualize or verify a produced model using an existing UML CASE tools.

Furthermore, a set of well-defined, automated and reusable model transformation patterns are proposed. We argue that using these patterns will simplify both the model development task and the task of specifying model transformations. We provide a set of model to model transformation using the proposed transformation patterns as templates. These reusable model transformations can be used to map different models to each other while they preserve the consistency between these models. Due to the fact that performing these model transformations by hand can be quite a time consuming and error-prone task, these automated model transformations improve developer productivity and reduce human error. Additionally, these executable transformations help developers to reduce the effort of software maintenance activities such as reverse engineering and refactoring.

We propose different set of transformations to map the PIM model to the PSM model. Most existing MDA tools provide only model-to-code transformations, we believe that providing an intermediate model (PSM) before generating the code makes the transformations more modular and maintainable. Also, it is possible to transform

the PIM to a non-aspect oriented environment to meet the stakeholders' needs. Furthermore, intermediate models can be used for optimization and tuning, or debugging purposes. We provide automation and tool support through an Eclipse plug-in and we demonstrate the effectiveness of our approach through the case studies.

The main limitation of this project is that it cannot work with codes that are written using programming languages while the majority of software systems continue to be developed by writing code using programming languages without any modeling. Hence, we need a set of transformations to transform code into the PSM model.

In the future, we plan to use the internal structures provided by the QVT engine [34] to handle change propagation by providing traceability links between different model elements. Once an element is modified, changes have to be made in dependent components to preserve the correctness of the system. We can traverse each model elements using these links to identify the affected elements.

Additionally, further works may concentrate on proposing new transformation patterns and completing the proposed transformation specifications to be more accurate and completely automated.

Bibliography

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

[2] J. Torres A. M. Reina. Weaving AspectJ aspects by means of transformations. In *The First Workshop on Models and Aspects- Handling Crosscutting Concerns in MDSD at the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, 2005.

[3] N. Albunni and M. Petridis. Using UML for modeling crosscutting concerns in aspect-oriented software engineering. In *Proceedings of the 3rd International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA)*, 2008.

[4] Omar Aldawud, Tzilla Elrad, and Atef Bader. UML profile for aspect-oriented software development. In *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling (AOM)*, 2003.

[5] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. *In AOSD workshop on Software Engineering Properties of Languages*

for Aspect Technologies (SPLAT), 2004.

[6] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, pages 73–87, New York, NY, USA, 2000. ACM.

[7] Alan Brown. An introduction to model driven architecture. Technical report, IBM, 2004.

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

[9] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Workshop on Generative Techniques in the Context of Model-Driven Architecture (OOPSLA)*, 2003.

[10] Edsger W. Dijkstra. On the role of scientific thought. *In Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1974.

[11] Tzilla Elrad, Robert E. Filman, and Atef Bader. Theme section on aspect-oriented programming. *Communications of ACM*, 44:29–32, 2001.

[12] Joerg Evermann. A meta-level specification and profile for AspectJ in UML. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM)*, 2007.

[13] David C. Fallside and Priscilla Walmsley. Xml schema part 0: Primer second edition. W3C Recommendation, October 2004.

[14] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the OOPSLA Workshop on Advanced Separation of Concerns*, 2000.

[15] Lidia Fuentes and Pablo Sánchez. Towards executable aspect-oriented UML models. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM)*, 2007.

[16] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[17] Maria-Eugenia Iacob, Maarten W. A. Steen, and Lex Heerink. Reusable model transformation patterns. In *Proceedings of the 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, pages 1–10, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Sheena R. Judson. Pattern-based model transformation. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 124–125, New York, NY, USA, 2003. ACM.

[19] José Uetanabara Júnior, Valter Vieira Camargo, and Christina Von Flach Chavez. UML-AOF: A profile for modeling aspect-oriented frameworks. In *Proceedings of the 13th International Workshop on Aspect-Oriented Modeling (AOM)*, 2009.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[21] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[22] Jörg Kienzle, Yang Yu, and Jie Xiong. On composition and reuse of aspects. In *Proceedings of the 2nd AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2003.

[23] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise.* 2003.

[24] Jana Koehler, Rainer Hauser, Shubir Kapoor, Fred Y. Wu, and Santhosh Kumaran. A model-driven transformation method. In *Proceedings of the 7th International Conference on Enterprise Distributed Object Computing (EDOC)*, pages 186 – 197, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[26] Ivan Kurtev. State of the art of QVT: A model transformation language standard. In *Applications of Graph Transformations with Industrial Relevance: Third International Symposium (AGTIVE).*

[27] Craig Larman. *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development (3rd edition).* Prentice Hall PTR, 2004.

[28] Ora Lassila and Ralph R. Swick. Resource description framework (RDF) model and syntax specification. Technical report, http://www.w3.org/TR/PR-rdf-syntax, 1998, (Retrieved: 09-08-2010).

[29] Eclipse M2M. Declarative QVT quick start. Technical report, http://www.eclipse.org/m2m/dqvt, (Retrieved: 09-08-2010).

[30] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[31] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[32] Marco Mosconi, Anis Charfi, Jaroslav Svacina, and Jan Wloka. Applying and evaluating AOM for platform independent behavioral UML models. In *Proceedings of the 7th AOSD International Workshop on Aspect-Oriented Modeling (AOM)*, 2008.

[33] OMG. UML 2.0 infrastructure final adopted specification. Technical report, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, 2003 (Retrieved: 09-08-2010).

[34] OMG. MOF QVT final adopted specification. Technical report, OMG, http://www.iist.unu.edu/ vs/wiki-files/MOFQVTSpec.pdf, 2005, (Retrieved: 09-08-2010).

[35] OMG. XML metadata interchange (XMI). Technical report, http://www.omg.org/cgi-bin/doc?formal/2007-12-02, 2007, (Retrieved: 09-08-2010).

[36] OMG. Object constraint language (OCL) 2.2. Technical report, http://www.omg.org/spec/OCL/, 2010, (Retrieved: 09-08-2010).

[37] OMG. MOF core specification. Technical report, http://www.omg.org/spec/MOF/2.0/, (Retrieved: 09-08-2010).

[38] Transformations Rfp Partners. Initial submission for MOF 2.0 Query / Views /, 2003.

[39] Adam Przybylek. Separation of crosscutting concerns at the design level: An Extension to the UML Metamodel. In *Proceedings of the 2nd International Multiconference on Computer Science and Information Technology (IMCSIT)*, 2008.

[40] Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML(TM)*. Cambridge University Press, New York, NY, USA, 2004.

[41] A.M Reina, J. Torres, and M Toro. Towards developing generic solutions with aspects. In *In proceedings of the workshop on Aspect Oriented Modeling (AOM)*, Lisbon, Portugal, 2004.

[42] Julia Reznik, Tom Ritter, Rudolf Schreiner, and Ulrich Lang. Model driven development of security aspects. *Electron. Notes Theor. Comput. Sci.*, 163(2):65–79, 2007.

[43] Julia Reznik, Tom Ritter, Rudolf Schreiner, and Ulrich Lang. Model driven development of security aspects. *Electronic Notes in Theoretical Computer Science*, 163(2):65–79, 2007.

[44] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. *SIGSOFT Softw. Eng. Notes*, 29(6):147–158, 2004.

[45] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[46] Bran Selic. A systematic approach to domain-specific language design using UML. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2007.

[47] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[48] Arnor Solberg, Devon Simmonds, Raghu Reddy, Sudipto Ghosh, and Robert France. Using aspect oriented techniques to support separation of concerns in model driven development. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 121–126, Washington, DC, USA, 2005. IEEE Computer Society.

[49] Guy St-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a model interchange format: The SPOOL case study. In *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS)*, 2000.

[50] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[51] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Designing aspect-oriented crosscutting in UML. In *Proceedings of Proceedings of the Aspect-oriented sofwtare development UML workshop (AOSD-UML)*, 2002.

[52] Bedir Tekinerdogan, Mehmet Aksit, and Francis Henninger. Impact of evolution of concerns in the model-driven architecture design approach. *Electron. Notes Theor. Comput. Sci.*, 163(2):45–64, 2007.

[53] Matthias Urban and Olaf Spinczyk. AspectC++ language reference, 2006.

[54] Dean Wampler. The role of aspect-oriented programming in OMG's model-driven architecture. *Aspect Programming, Inc.*, 2003.

[55] K. Wong and H. Mller. Rigi users manual, version 5.4.4. Technical report, University of Victoria, Victoria, Canada, ftp://ftp.rigi.csc.uvic.calpub/, 1998, (Retrieved: 09-08-2010).

[56] Xiao-Cong Zhou, Chang Liu, Yan-Tao Niu, and Tai-Zong Lai. Towards a framework of aspect-oriented modeling with UML. In *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology (ISCSCT)*, 2008.