
# Analysis of Crosscutting Concerns in QVT-based Model Transformations

**H.Q. Nguyen**

**July 2006**

**Master of Science Thesis**

# Analysis of Crosscutting Concerns in QVT-based Model Transformations

**by**

**H.Q. Nguyen**

**July 2006**
**Enschede**

| | |
|---|---|
| **Graduation Committee:** | **dr. ir. K.G. van den Berg** |
| | **dr. ir. B. Tekinerdogan** |
| | **prof. dr. ir. M. Aksit** |
| **Chair:** | **Software Engineering** |
| **Department:** | **Electrical Engineering, Mathematics and Computer Science** |
| **University:** | **University of Twente** |

# Abstract

The Model Driven Architecture (MDA) framework aims at the idea of separating the specification of a system from the details of the way that system uses the capabilities of its platform. It intends to reach this goal by defining models and transformations between models which are both based on metamodels. To provide a language for model transformations, a Query, Views and Transformations (QVT) specification has been proposed as a combination from several submissions in response to the QVT Request for Proposal by the Object Management Group (OMG).

Separation of concerns is a basic principle in software engineering. The principle states that a given problem involves different kinds of concerns, which should be identified and modularized to cope with complexity and to achieve quality factors such as robustness, adaptability, maintainability and reusability. However, crosscutting concerns like security, persistence and distribution cause difficulties to the achievement of this principle. Aspect-Oriented Software Development (AOSD) emerges to address this problem in order to improve modularity and maintainability of systems, and is dealing with the problem of crosscutting concerns. AOSD provides systematic techniques for the identification, modularization, representation and composition of crosscutting concerns.

Despite its importance, the problem of crosscutting concerns has not been extensively considered in MDA. The consequence is that it is not known what problems crosscutting concerns cause to QVT model transformations, and how QVT model transformations can be used to deal with the problem of crosscutting concerns. To analyze this problem, we make a detailed review of the concepts related to QVT model transformations and crosscutting concerns. We then choose a formal definition of crosscutting which is based on dependency graphs between elements of a source and a target.

Traceability in model transformations could be used to derive this kind of dependency graphs. We propose a method to derive dependency graphs for transformation rules at both metamodel and model levels. This method is based on the specification of the rules written in the Relations and Core languages of the QVT specification and the related tracing model. It also distinguishes two different types of mappings in these dependency graphs: direct and indirect mappings. These dependency graphs help to identify problems that crosscutting concerns cause to model transformations. Some of the problems relate to properties of model transformations, such as transformation rule interaction and execution order of transformation rules. Others are general problems, such as the complexity and change impact on model transformations.

It is also concluded that these dependency graphs could be used to identify crosscutting concerns. However, there is a tendency that every element crosscuts other elements as the dependency graphs become very large and complex in a real-life model transformation. It is recommended that these dependency graphs should be further processed in order to identify crosscutting concerns more effectively. Some directions are to take into account the number of mappings between and the degree of granularity of the source and target elements.

# Acknowledgements

This Master of Science thesis could not be completed without the contribution and support of many people. I would like to thank all of them.

First of all, I would like to thank my supervisors, Klaas van den Berg and Bedir Tekinerdogan, for their guidance and support. Their knowledge and dedication are the foundation of this thesis.

A special thanks to Jan Schut, Belinda Jaarsma-Knol and Brenda Benders. They give me all kinds of support before and during the time I attend this master program.

I wish to thank all of my friends in the small Vietnamese community in Enschede. They help me to overcome all the difficulties and enjoy with me for all the happiness.

Finally, I would like to thank my wife Huong and my little daughter Ha Linh for their patience and encouragement.

# Table of Contents

# Table of Figures

# Table of Tables

# List of Abbreviations

| | |
|---|---|
| AO | Aspect Orientation |
| AOP | Aspect-Oriented Programming |
| AOSD | Aspect-Oriented Software Development |
| CFVS | Concurrent File Versioning System |
| COSMOS | COncern-Space MOdeling Schema |
| CWM | Common Warehouse Metamodel |
| EJB | Enterprise Java Bean |
| EMOF | Essential Meta-Object Facility |
| GROOVE | GRaphs for Object-Oriented VErification |
| LHS/RHS | Left-Hand Side/Right-Hand Side |
| MDA | Model Driven Architecture |
| MDE | Model Driven Engineering |
| MOF | Meta-Object Facility |
| OCL | Object Constraint Language |
| OMG | Object Management Group |
| PIM | Platform Independent Model |
| PSM | Platform Specific Model |
| QVT | Query, Views and Transformations |
| RDBMS | Relational Database Management System |
| RFP | Request for Proposal |
| SCM | Software Configuration Management |
| SQL | Structured Query Language |
| TRESE | Twente Research and Education on Software Engineering |
| UML | Unified Modeling Language |
| XMI | XML Metadata Interchange |
| XML | Extensible Markup Language |

# 1 Introduction

## 1.1 Context

The Model-Driven Architecture (MDA) is a set of standards proposed by the Object Management Group (OMG) for software development based on models and model transformations. MDA helps to separate the specification of a system from the details of the way that system uses the capabilities of its platform. [17]

MDA provides an approach, and enables tools for:

- Specifying a system independent of the platform that supports it
- Specifying the platforms
- Choosing a particular platform for the system, and
- Transforming the system specification to the chosen platform.

The goals of MDA and its model transformations are portability, interoperability and reusability through architectural separation of concerns. [17]

In MDA, models are the primary artifacts. Abstract models are transformed to more concrete models and eventually to platform specific models from which executable models can be generated. Automatic transformations play a key role. OMG issued a Request for Proposal (RFP) for MOF 2.0 Query, Views and Transformations (QVT) to address the technology part of the OMG MOF 2.0 in manipulation of MOF (Meta-Object Facility) models.

Another principle in software engineering is the separation of concerns. A concern is commonly considered as a matter of interest in a software application. Many different kinds of concerns are relevant to different stakeholders at different stages in a software lifecycle [2]. The principle states that a given problem involves different kinds of concerns, which should be identified and modularized to cope with complexity and to achieve quality factors such as robustness, adaptability, maintainability and reusability [8][30]. However, there are concerns such as security, persistence and distribution that do not fit naturally into a single program module, or even several closely related program modules. This type of behavior is termed crosscutting because it cuts across the dominant decomposition of responsibility of a given programming model [8][30]. These concerns are called crosscutting concerns.

Aspect-Oriented Programming (AOP) is a new programming paradigm to provide strategy for dealing with the problem of crosscutting concerns at the programming level by providing new language constructs. AOP is further developed into Aspect-Oriented Software Development (AOSD). AOSD generalizes AOP to other phases of software development. The idea is to implement individual concerns and then combine these implementations to form the final system.[8]

There have been proposals to apply AOSD in the MDA framework in order to cope with the problem of crosscutting concerns. Some solutions propose a general framework for aspect-oriented modeling, such as the Concern-Space Modeling Schema (COSMOS) by Sutton and Rouvellou [8][26], while others recommend specific elements to include in the UML standard, such as the approach for generic aspect-oriented design with Theme/UML by Clarke and Walker [8], or expressing aspects using UML behavioral and structural diagrams by Elrad et al. [8].

## 1.2 Problem and Approach

Despite these proposals, the problem of crosscutting concerns has not been extensively considered in MDA. The consequence is that it is not known whether the QVT model transformations, a key element of MDA, are able to effectively deal with existing and future AOSD techniques which have been proposed to be applied in the MDA framework.

In order to analyze the problem of crosscutting concerns in QVT model transformations, this study aims at answering the following questions:

- How can crosscutting concerns be identified in QVT model transformations?
- What are the problems of crosscutting concerns in the definition and execution of QVT model transformations?

The approach of this research is to use traceability to analyze the relationship between crosscutting concerns and model transformations (See Figure 1). Traceability is a fundamental property of model transformations and implementation of traceability is one of the requirements of the OMG's RFP. Trace dependencies can also be used to identify crosscutting concerns as discussed in subsequent chapters.

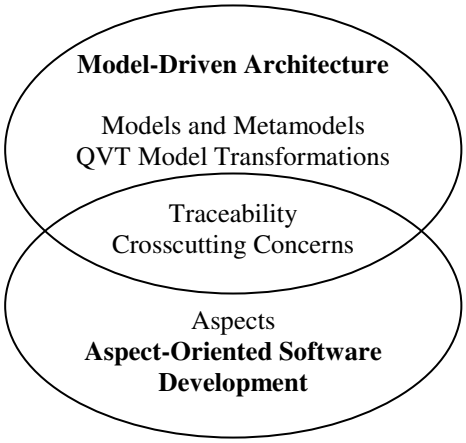



***Figure 1: Topics in MDA and AOSD covered in this thesis***

A case study is used to illustrate the related concepts and to provide needed facts for the analysis. In this study, the example of a Concurrent File Versioning System from the Master thesis of Henninger [9] is reused. The design model and transformation rules to transform this design model to the implementation models are defined. Trace information from this transformation is used to derive the dependency graphs at metamodel and model levels. These dependency graphs are the basis for analyzing crosscutting concerns on model transformations in this case study.

## 1.3 Contribution of the Thesis

This thesis makes several contributions.

- The study provides a thorough analysis of the QVT language, especially its declarative languages Relations and Core. This language has just been proposed and it is still on the way to become standardized. Thus a detailed explanation through an actual case study is very helpful to understand the concepts used by the language and to evaluate its effectiveness to deal with model transformations.
- The study uses traceability as a common property to analyze of the relation between crosscutting concerns and QVT model transformations. This analysis is helpful in identifying crosscutting concerns in the development of a software application. It is also used to identify several fundamental characteristics of QVT model transformations such as transformation rule interaction and execution order of transformation rules.
- The study extends and proposes a framework to derive dependency graphs from transformation rules and the tracing information. These dependency graphs are derived as mappings between source and target elements in a transformation at both metamodel and model levels. The mappings could be direct and indirect. Formal definitions of dependency graphs for transformation rules and direct and indirect mappings are provided.

## 1.4 Organization of the Thesis

The remaining of this thesis is organized as follows:

- **Chapter 2** introduces Model Driven Engineering and model transformations. In this chapter, overview of Model Driven Engineering and Model Driven Architecture is presented, followed by a detailed explanation of the QVT transformation language which is proposed by the QVT Merge Group in response to the Request for Proposal for MOF 2.0 QVT by the OMG.
- **Chapter 3** provides definitions for various concepts of scattering, tangling and crosscutting. These definitions are the basis for analyzing the impact of crosscutting concerns on QVT-based model transformations.
- **Chapter 4** presents the case study – the Concurrent File Versioning System. This case study is reused from the Master thesis of Henninger. The UML design model and some transformation rules to transform this model to Relational and Java implementation models are defined. An overview of the tools for this transformation is also provided.
- **Chapter 5** analyzes the crosscutting concerns in the case study and how they impact the model transformation. The analysis uses the dependency graphs derived from the definition of the transformation rules (at metamodel level) and from the tracing information of the transformation execution (at model level)
- **Chapter 6** finally provides the conclusions of this analysis and discusses the results and the future study.

# 2 QVT Model Transformations

This chapter describes a Query, Views and Transformations (QVT) language for model transformations. Section 2.1 briefly introduces the Model Driven Architecture (MDA) as adopted by the Object Management Group (OMG) and the evolution of the QVT language. Section 2.2 compares the QVT language with Czarnecki's classification of model transformation approaches [7] and Section 2.3 evaluates how the language complies with the requirements of the OMG's Request for Proposal [22]. Finally, Section 2.4 gives an explanation of the language based on examples of transformation rules written in two of its sub-languages.

## *2.1 Model Driven Engineering*

### 2.1.1 Model Driven Architecture

The Model Driven Architecture (MDA) is a software development framework adopted by the Object Management Group (OMG). The goal of MDA is to solve several problems that are common in software development processes [9][11][12][17]:

- *Portability*: whenever new technologies emerge, companies are forced to port their software systems to a new environment. The constant changes in technologies may create a problem with portability which may require significant efforts.
- *Productivity*: current software development practices have the problem of productivity due to the fact that we have to spend too much effort on low-level design and coding. The maintenance and understanding of code is difficult for large software systems.
- *Interoperability*: software systems usually consist of multiple components which are built upon different technologies. Therefore, these components need to interoperate with each other.

There are two principles in MDA to cope with these problems. The first principle is to use modeling and models to develop software systems. The second principle is the separation of a system from the details of how the system is implemented via concrete technologies.

MDA classifies models into two classes based on their levels of abstraction, namely Platform Independent Models (PIM) and Platform Specific Models (PSM), which rely on the concept of platform. The MDA Guide [17] defines these classes and concept as follows:

- *Platform Independent Model*: is a view of a system from a platform independent viewpoint.
- *Platform Specific Model*: is a view of a system from a platform specific viewpoint.
- *Platform*: a platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

The development of a system based on the MDA approach starts from building a PIM of that system. The PIM is then transformed to one or more PSM's which use constructs provided by the chosen platforms. The PSM's are finally transformed to code.

The basic operation that is applied to models in this approach is the model transformation. The model transformation is the process of converting one model to another model of the same system. MDA aims at automating model transformations as much as possible by tools based on transformation specifications. Figure 2 shows the MDA transformation process as defined in the MDA Guide. In the typical case, the source and target models are the PIM and PSM, respectively, while the details of the chosen platform and parameters to the transformation are represented by the additional information.

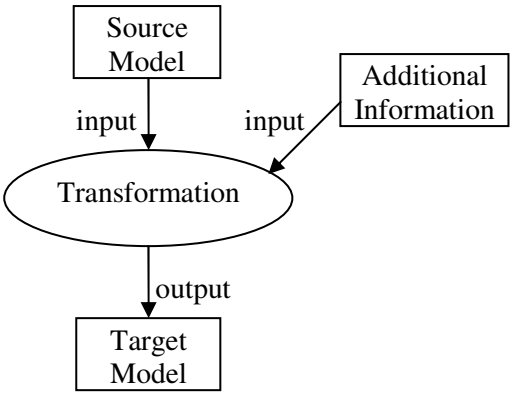



*Figure 2: The MDA Transformation Pattern [17]*

### 2.1.2 Model Driven Engineering

MDA lacks the notion of software development process. Model Driven Engineering (MDE) is the enhancement of MDA in this direction. Kurtev [12] discusses MDE on the basis of MDA by adding the notion of software development process and a modeling space for organizing models. There are also requirements for tools needed to perform operations on models in MDE.

The modeling space in MDE has multiple dimensions. The first dimension is the level of model abstraction, or degree of Abstractness and Concreteness. This dimension is also defined in MDA based on the classification of models into PIM and PSM, in which PIM is more abstract than PSM.

The second dimension comes from the distinction of models based on the subject areas they belong to. Different stakeholders of the software system may have different views focusing on a subset of features of that system. These views are reflected in different models. These subject areas correspond to concerns as defined in the Aspect-Oriented Software Development, such as security, persistence, and distribution.

The third dimension is concerned with organizational issues such as versioning and authorship over models.

It is also mentioned in [12] about the vision of MDE where MDA is just one possible instance of MDE implemented on a set of technologies proposed by OMG (MOF, UML, XMI, etc.). The concepts of model, metamodel and transformations are available in other technologies as well.

### 2.1.3 Query, Views and Transformations

The OMG's MDA is a software development approach which uses models as the primary artifacts. Abstract models are usually transformed to more concrete models and eventually to platform specific models from which executable artifacts are generated (code and configuration files).

In general, model transformations are the operations written in a transformation language to convert a source model to a target model. The transformation may use metamodels to define the instantiation of elements in the target model from elements in the source model. The MDA transformation pattern in Figure 2 could be extended with the use of metamodels as shown in Figure 3 below.

There are several significant benefits of model transformations. One of these benefits is reusability. During the development of a software product using the MDA approach, the software is modeled as models at different levels of abstraction. If there are changes to a model, these changes are propagated to other models. This process is usually repeated in a specific algorithm which may be implemented in a transformation language as transformation rules to be reused in the development of that software product as well as other ones.

Model transformations also provide the automation for many manual activities. This would help to increase the productivity and decrease the possibility of errors in software

development. For instance, the PIM model may be initially simple and abstracted out of unnecessary details. The designers may later apply design patterns to create more complicated PSM model for a specific platform. Transformation rules thus can be pre-implemented and attached to individual design patterns so that these platform specific models are generated automatically.

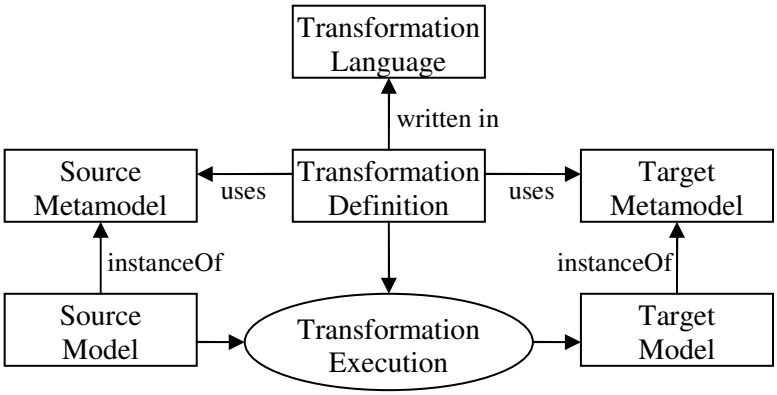



***Figure 3: The use of metamodels in a transformation definition [12]***

Because of these benefits, automatic transformations in MDA play a key role. It is important that transformations can be developed as efficiently as possible. In 2002, OMG issued a Request for Proposal (RFP) for MOF 2.0 Query, Views and Transformations (QVT) to address the technology part of the OMG MOF 2.0 in manipulation of MOF models [22].

In this RFP, three fundamental technology parts are mentioned, namely queries, views and transformations. These terms are defined as follows:

- *Query*: a query is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language.
- *View*: a view is a model that is completely derived from another model. A view cannot be modified separately from the model from which it is derived. The metamodel of the view is not necessary to be the same as the metamodel of the source model. A query is a restricted kind of view. Views are generated via transformations.
- *Transformation*: a transformation generates a target model from a source model. Transformations may lead to dependent or independent models. A transformation may be top-down in which case the target model is not modified after generation. Transformations may be one-way (unidirectional) or two-way (bidirectional).

In response to the RFP, 8 proposals were submitted to OMG in October 2002. Gardner et al. [15] evaluated these proposals and found that many submissions were incomplete and so unclearly formulated. During the year 2003 and 2004, these submissions were revised and converged. As of January 2005, two submissions remained under consideration. Finally, in March 2005, these two submissions were merged again to provide the joint 3rd revised submission by the QVT Merge Group.

As the only remaining submission, the proposal by the QVT Merge Group, with further refinement, will be considered to be the specification for the QVT language. In this assignment, this proposal will be used as the QVT language for the problem under consideration. The remaining sections discuss how this QVT language meets the requirements provided in the RFP and various concepts used by the language.

## 2.2 Classification of the QVT Language

Czarnecki et al. [7] proposes a feature model to compare different model transformation approaches and offers a survey and categorization of a number of existing approaches which are published in papers and submitted in response to the OMG's QVT RFP [22]. The feature diagrams are presented in the following subsections.

### 2.2.1 Main Features of Model Transformations

The main feature diagram of model transformations is shown in Figure 4. Model transformations have optional (marked with white bullets) and mandatory (marked with solid bullets) features, each of which is elaborated in the following subsections.

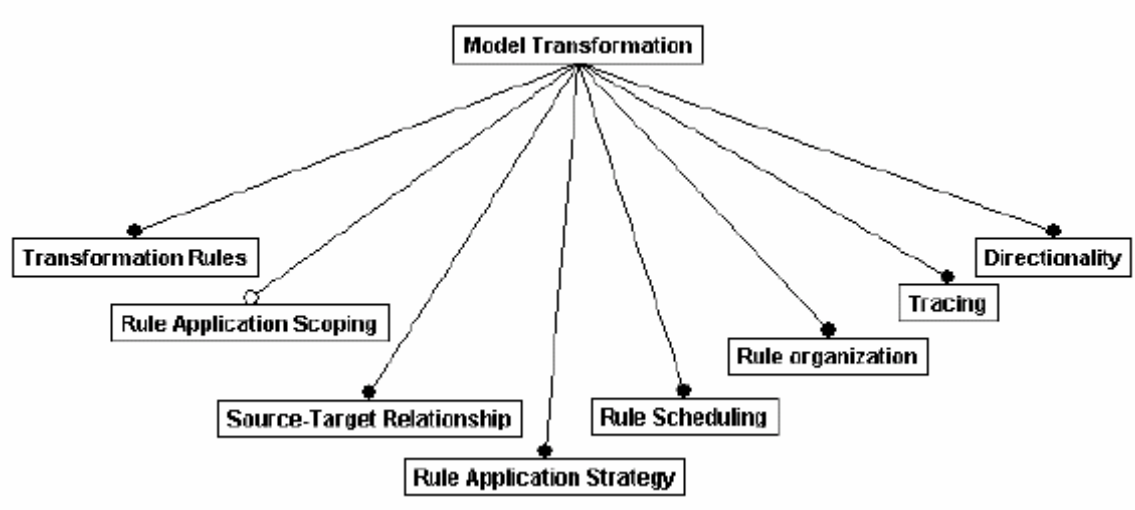



*Figure 4: Top-level features of model transformations [7]*

### 2.2.2 Transformation Rules

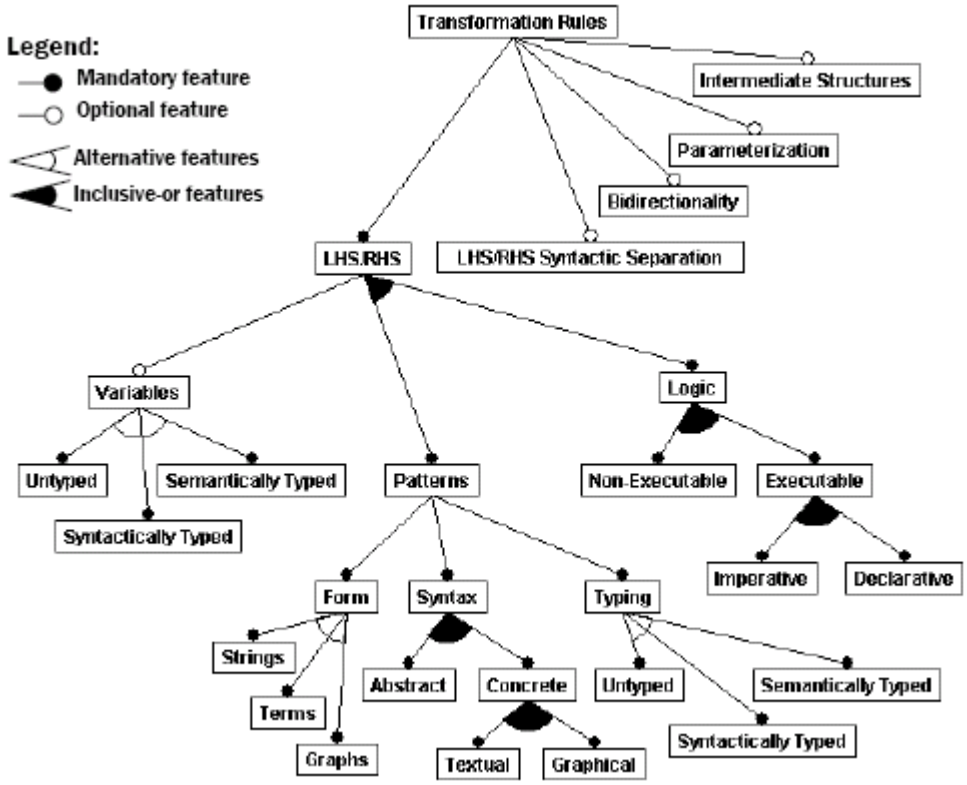



*Figure 5: Features of transformation rules [7]*

Figure 5 presents features of transformation rules. In this QVT language, the concept of source and target models (or *LHS/RHS*) is generalized to multi-directional transformations in which multiple models can be specified. All features of *variables*, *patterns* and *logic* can be used to represent these models.

There is a *syntactic separation* between models involved in the transformation. For the Relations and Core languages, *multi-directionality* (which is stronger than *bi-directionality*) can be achieved by specifying any model in transformation rules as the target model, whereas the remaining are the sources. *Rule parameterization* can also be defined during transformation definition and execution.

### 2.2.3 Rule Application Scoping

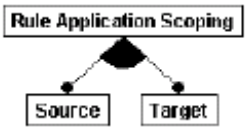



***Figure 6: Features of rule application scoping [7]***

Rule application scoping allows restricting the parts of a model that participates in the transformation for performance reasons. This feature is not present in this QVT specification.

### 2.2.4 Relationship between source and target models

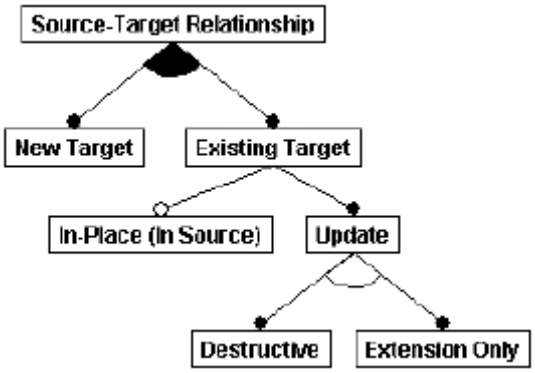



***Figure 7: Features of relationship between source and target models [7]***

In this QVT specification, the target model may be empty, but it has to exist (*Existing Update*). Elements in the target model may be created, deleted or modified in order to make the relation hold (*Update*). The target model may also be the same as the source model; i.e. *in-place* update, however, additional evaluation steps may be needed.

### 2.2.5 Rule Application Strategy

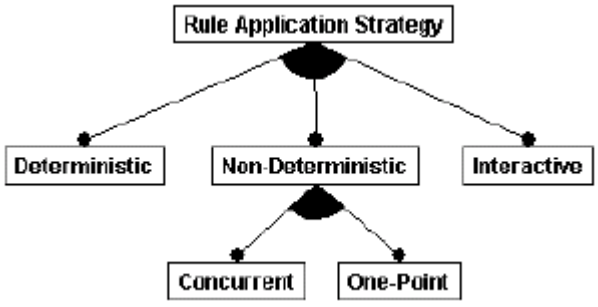



***Figure 8: Features of rule application strategy [7]***

This feature classifies transformation approaches by the order with which a specific rule is applied to multiple matches found in the source models. However, it has not been found in the submission document which application strategy the specification uses.

### 2.2.6 Rule Scheduling

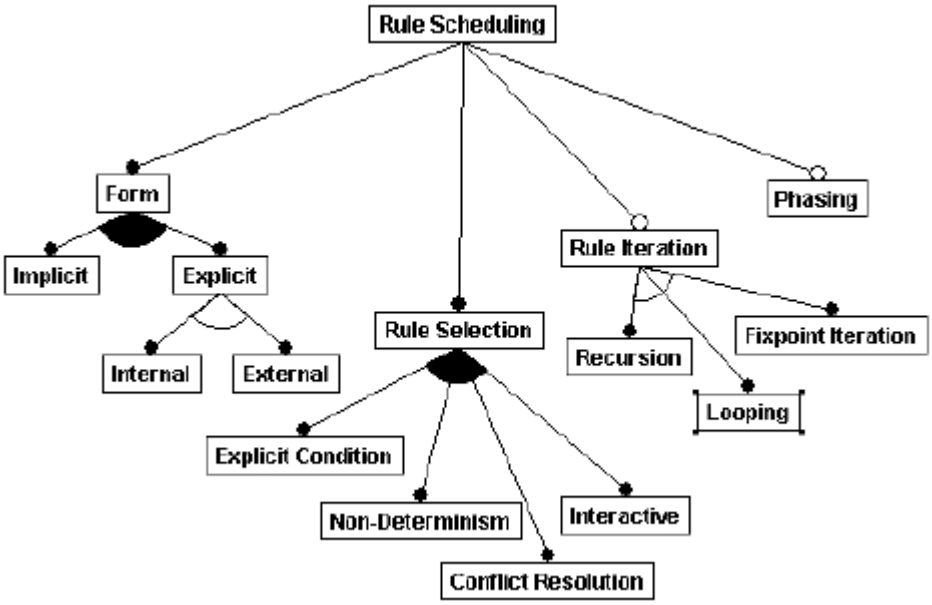



***Figure 9: Features of rule scheduling [7]***

Multiple transformation rules can be specified to transform source models to target models. The *Rule Scheduling* feature classifies the order in which individual rules are applied. In this specification, there are two types of rules: top-level and others. For example, the Relations language uses the keyword `top` to mark the top level relations. The top-level rules are executed directly by the transformation engine, other rules are invoked indirectly from the top-level rules, such as in the `where` and `when` clauses. However, it is not clarified which order the top-level rules are executed.

### 2.2.7 Rule Organization

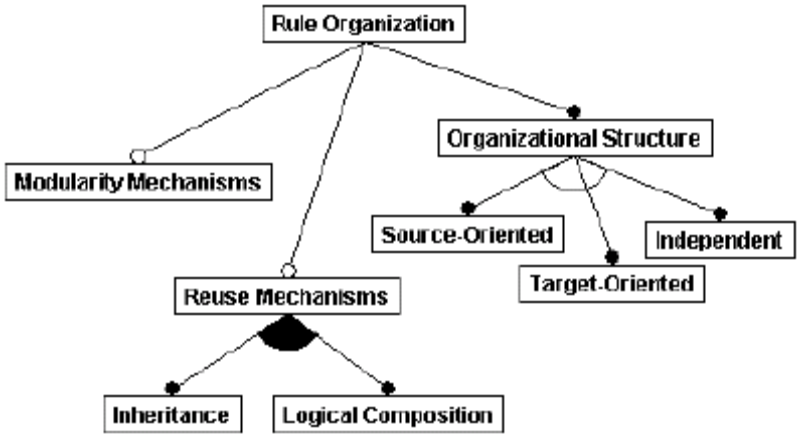



***Figure 10: Features of rule organization [7]***

According to the submission document, transformation rules are organized in *modules*. The submission also states that it supports *reuse mechanism* through the use of MOF polymorphism; however, no examples have been found for this. *Logical composition* is also used to compose multiple transformation rules. Finally, the rules are organized *independently* from the source and target models.

### 2.2.8 Traceability

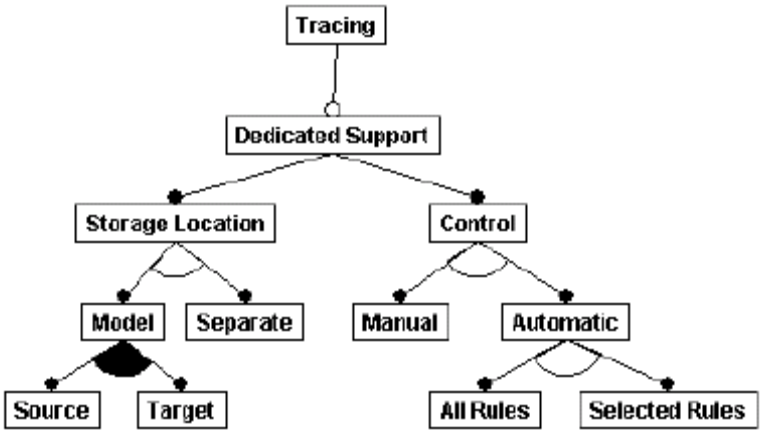



***Figure 11: Features of traceability [7]***

Trace classes and their instances are implicitly generated in the Relations language and Operational Mappings to maintain the traceability between model elements in a transformation execution. However, they have to be created explicitly in the Core language. The *location* of these instances is not described clearly in the document; however, it can be inferred that these instances are stored *separately* from the source and target models and are maintained by the transformation engine.

### 2.2.9 Directionality

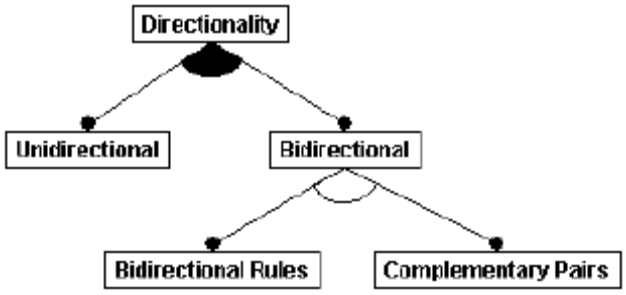



***Figure 12: Features of directionality [7]***

*Multi-directionality* is supported in the declarative languages Relations and Core. Multiple candidate models can be specified in a *multi-directional transformation rule* and any one of them can be nominated as the target model in a transformation execution. However, the imperative Operation Mappings language requires a separate rule for each direction of the transformation (*complementary pairs*).

## *2.3 Requirements Compliance of the QVT Language*

The Request for Proposal for the QVT language [22] provides general and specific requirements; specific requirements in turns are divided into mandatory and optional requirements. This section extracts specific requirements of the RFP and, based on the submission document, discusses the resolution of this QVT language to these requirements.

### 2.3.1 Mandatory Requirements

*1. Proposals shall define a language for querying models. The query language shall facilitate ad-hoc queries for selection and filtering of model elements, as well as for the selection of model elements that are the source of a transformation.*

This submission uses OCL 2.0 as its query language. The QVT models also introduce some subtypes of OCL metamodel elements. No explicit support for ad-hoc queries is provided in this submission.

*2. Proposals shall define a language for transformation definitions. Transformation definitions shall describe relationships between a source MOF metamodel S, and a target MOF metamodel T, which can be used to generate a target model instance conforming to T from a source model instance conforming to S. The source and target metamodels may be the same metamodel.*

In the Relations and Core languages, multi-directional transformation definitions can be specified. These languages allow the nomination of a model at runtime which defines the direction the transformation will execute in. The Operational Mappings specifications always have a direction defined in terms of a target model.

*3. The abstract syntax for transformation, view and query definition languages shall be defined as MOF (version 2.0) metamodels.*

All packages in this QVT language are defined using EMOF, a subset of MOF 2.0, and extend the MOF 2.0 and OCL 2.0 specifications.

*4. The transformation definition language shall be capable of expressing all information required to generate a target model from a source model automatically.*

By using the Relations, Core and Operational Mappings languages, transformation definitions are capable of expressing all information required to generate a target model from a source model automatically. An optional feature to define some parts of the transformation is to use Operation black-box implementations; however, the portability of transformation will be not guaranteed.

*5. The transformation definition language shall enable the creation of a view of a metamodel.*

This specification does not support views.

*6. The transformation definition language shall be declarative in order to support transformation execution with the following characteristic:*

- *Incremental changes in a source model may be transformed into changes in a target model immediately.*

This mode of execution is supported by the Relations and Core languages. The imperative approach is also supported by the Operational Mappings

*7. All mechanisms specified in Proposals shall operate on model instances of metamodels defined using MOF version 2.0.*

The specification supports this requirement.

### 2.3.2 Optional Requirements

*1. Proposals may support transformation definitions that can be executed in two directions. There are two possible approaches:*

- *Transformations are defined symmetrically, in contrast to transformations that are defined from source to target.*
- *Two transformation definitions are defined where one is the inverse of the other.*

Both approaches are supported in this specification. The Relations and Core languages support the former, while the Operational Mappings supports the latter.

*2. Proposals may support traceability of transformation executions made between source and target model elements.*

Trace classes are implicitly created in the Relations language to trace executions of Relations and Operational Mappings transformations. In the Core language, trace classes are explicitly created and matched by transformations.

*3. Proposals may support mechanisms for reusing and extending generic transformation definitions. For example: Proposals may support generic definitions of transformations between general metaclasses that are automatically valid for all specialized metaclasses. This may include the overriding of the transformations defined on base metaclasses. Another solution could be support for transformation templates or patterns.*

This specification supports this kind of specialization, as transformations are defined using MOF polymorphism.

*4. Proposals may support transactional transformation definitions in which parts of a transformation definition are identified as suitable for commit or rollback during execution.*

This specification does not deal with transactions.

*5. Proposals may support the use of additional data, not contained in the source model, as input to the transformation definition, in order to generate a target model. In addition proposals may allow for the definition of default values for this data.*

MOF Tags may be included for name/value pairs during transformation definition. For data required to execute a transformation, it is allowable to define properties which must be given appropriately typed values at runtime.

*6. Proposals may support the execution of transformation definitions where the target model is the same as the source model; i.e. allow transformation definitions to define updates to existing models. For example a transformation definition may describe how to calculate values for derived model elements.*

In this specification, a transformation is considered in-place when its source and target directions are bound to the same model at runtime. Some guidance is given in the specification for this kind of transformation.

## 2.4 Description of the QVT Language

This section describes the QVT specification by first introducing the overview of the language and then explaining the main concepts used in its declarative sub-languages Relations and Core which will be used in the case study.

### 2.4.1 Overview of the QVT Language

The QVT is a hybrid of declarative/imperative approach, with the declarative part being split into two-level architecture.[23]

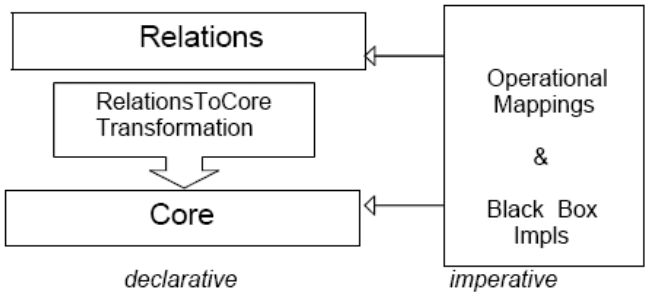



***Figure 13: Relationship between QVT metamodels [23]***

**Two-level Declarative Architecture**

The user-friendly Relations metamodel and language allows a declarative specification of transformation rules as relationships between MOF models. It supports complex object pattern matching expressions to locate and create and/or update elements of the models involved in a transformation. The transformation rules written in the Relations language implicitly create trace classes and their instances to record the mappings between source and target elements occurred during a transformation execution.

The Core metamodel and language supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. Besides, trace classes

and their instances must be explicitly defined and created, and are not deduced from the transformation definition, as with the Relations language. The Core language is as powerful as the Relations language, and because of its relative simplicity, its semantics can be defined more simply, although the transformation is described more verbosely.

The semantics of a Relations model is defined by a transformation to a trace model and a Core model with equivalent semantics. Thus Relations and Core models are considered as two models with the same semantics at different levels of abstraction.

Another view of the relationship between the Relations and Core languages is the analogy to the Java architecture, where the Core language is like the Java Byte Code and the Relations language plays the role of the Java language. The transformation from Relations to Core is like the specification of a Java compiler [23].

**Imperative Implementations**

The imperative part has two mechanisms to invoke the imperative implementations of transformations from Relations or Core: one standard language, Operational Mappings, and a non-standard Black-box MOF Operation Implementations. Each relation defines a class which will be instantiated to trace between model elements being transformed and it has a one-to-one mapping to an Operation signature that the Operational Mappings or Black-box implements.

The Operational Mappings language is specified in a standard way to provide imperative implementations. It populates the same trace models as the Relations language. It provides OCL extension with side effects which allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers.

For Black-box implementations, MOF Operations may be derived from Relations to "plug-in" any implementation of a MOF Operation with the same signature. This allows more flexibility to transformations. However, it is also dangerous. The transformation is less portable and the implementation has access to object references in models and may do arbitrary things to those objects.

**Package dependencies**

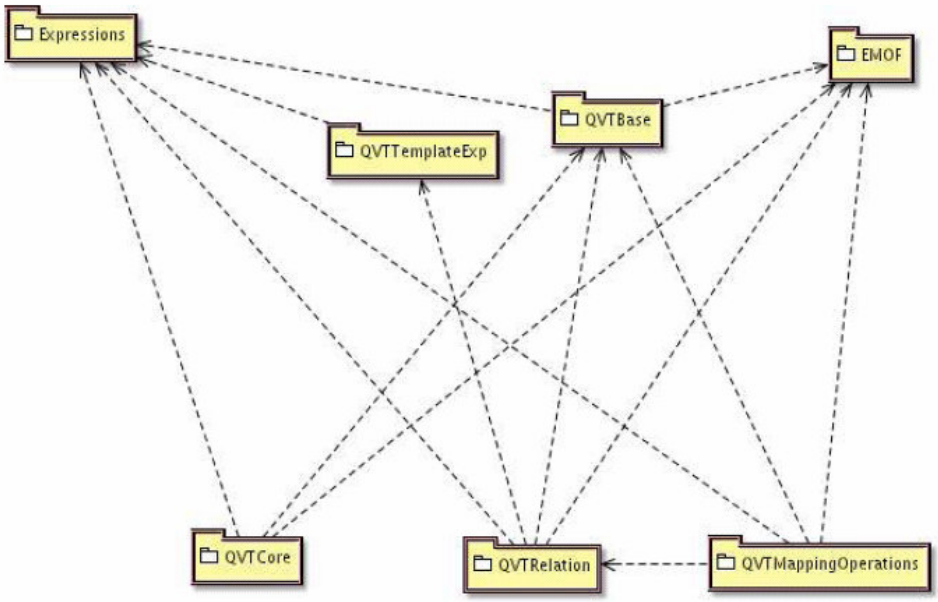



***Figure 14: Dependencies between Packages defined in the QVT specification [23]***

The specification defines three main packages, one for each language: `QVTCore`, `QVTRelation` and `QVTMappingOperations`. All packages depend on the `QVTBase` package which defines a common structure for transformations. Besides, the `QVTRelation` package uses Template Pattern Expressions defined in the `QVTTemplateExp` package. `QVTMappingOperations` extends the `QVTRelation` because it uses the same

framework for traces defined in that package. Finally, All QVT packages depend on the `Expressions` package from OCL 2.0, and on `EMOF`. Figure 14 presents these packages and dependencies between them.

Because the Relations and Core languages have the same semantics at different levels of abstraction, and the Operational Mappings and Black-box Implementations are just to provide the imperative implementation based on the Relations framework, the next subsections will introduce the Relations and Core languages in order to explain the concepts used in the specification for this QVT language. The case study in the following chapters will use the Relations and Core languages to investigate the problem under consideration.

### 2.4.2 The Relations Language

The example of a transformation from a UML model to a Relational model which is used to create database schemas is usually used to demonstrate the concepts and capabilities of QVT languages. This transformation is as follows: each package in the UML model corresponds to a schema in the Relational model, and all persistent classes and their attributes belonging to each package are used to create tables and their columns in the corresponding schema. Finally, foreign keys in the Relational model are created based on associations between classes in the UML model. Table 1 shows part of this transformation written in the Relations language and is used to explain the related concepts.

```
transformation umlRdbms(uml uses SimpleUML, rdbms uses SimpleRDBMS) {
key Table (schema, name);
key Column (name, owner);
key Key (name, owner);

top relation ClassToTable {
    cn, prefix: String;
    checkonly domain uml c:Class {
        namespace = p:Package {},
        kind = 'persistent',
        name = cn
    };
    enforce domain rdbms t:Table {
        schema = s:Schema {},
        name = cn,
        column = cl:Column {name = cn + '_tid', type = 'NUMBER'} ,
        key = k:Key {name = cn + '_pk', column = cl}
    };
    when {
        PackageToSchema(p,s);
    }
    where {
        prefix = '';
        AttributeToColumn(c, t, prefix);
    }
}
...
```

***Table 1: UML-to-Relational transformation definition in Relations (partly) [23]***

**Transformations and typed models**

A transformation consists of a set of *relations* (or transformation rules) that must hold between elements of a set of candidate models. These models are named and the types of elements they can contain are restricted to those defined in the packages or metamodels these models refer to. As the example shows, the transformation `umlRdbms` is defined as a set of relations, such as `ClassToTable`, `PackageToSchema` and `AttributeToColumn`, between elements from `uml` and `rdbms` models which are instances of `SimpleUML` and `SimpleRDBMS` metamodels.

At execution time, the *direction* of this transformation is identified by specifying one of the models, i.e. either `uml` or `rdbms`, as the target. The transformation execution is performed by making the relations hold by modifying only the target model. Some relations only check whether a relationship holds between candidate models and the transformation execution would either confirm this or report errors when it does not.

**Relations and domains**

A *relation*, or a transformation rule, consists of two or more *domains* and a pair of constraints known as the *guard* clause (or *when* clause) and the *where* clause. Each *domain* includes a candidate model, its type, and a set of *patterns* which can be viewed as a graph of object nodes, their properties and association links between them. An alternative view of pattern is a set of variables and a set of constraints that model elements bound to these variables must satisfy in order to be qualified as a valid binding. The domain pattern can be considered as a template for objects and their properties that must be located, modified or created in order to satisfy the relation.

The relation `ClassToTable` in the example defines two domains for `uml` and `rdbms` candidate models of types `SimpleUML` and `SimpleRDBMS`, respectively. The domain for `uml` contains a pattern used to locate and bind elements from the `uml` model to its variables `c`, `p` and `cn`. The meaning of this simple pattern is to locate all classes (bound to `c`) in the `uml` model belonging to a given package (bound to `p`) and having its `kind` property to be a literal value `'persistent'`; i.e. classes to be transformed to tables.

**Check and enforce**

A relation domain may be marked as either *checked* or *enforced*. A checked domain, when specified as the target in a transformation execution, is only checked to see whether there is at least a valid match in the model which satisfies the relationship with any match in the source models. For an enforced domain, when being specified as the target model in a transformation execution, if the relation does not hold, its elements are modified, created or deleted in order to satisfy the relationship.

In the above example, when the enforced `rdbms` model declared in the `ClassToTable` relation is specified as the target, if there is no table in the `rdbms` model corresponding to a matched class in the `uml` model, a table is created to make the relationship hold. On the other hand, when the transformation is executed in the checked `uml` model direction, the transformation execution for the relation `ClassToTable` only checks whether there is a corresponding class in the `uml` model for each table in the `rdbms` model.

**When and where clauses**

A relation is constrained by two predicates. The *when* clause ensures that the relation is only checked or enforced when the clause's predicate is evaluated to true. When the when clause does not hold, the relation is ignored; thus the clause is also called the *guard* clause. The *where* clause specifies the condition that must be satisfied by all model elements participating in a relationship; it is usually used as a way to extend the relation to other relations defined elsewhere.

The when clause in the example ensures that the relation is not maintained for all matched classes in the `uml` model and all matched tables in the `rdbms` model, but only for classes and tables in a pair of corresponding package and schema via the relation `PackageToSchema`. In other words, the when clause with the relation `PackageToSchema` constrains the binding of elements to variables of the domain patterns.

When a valid binding of elements (classes and tables in a pair of corresponding package and schema) to variables is found, the relation is checked or enforced further by the relation `AttributeToColumn` in the where clause which ensures that the relationship between the class's attributes and the table's columns also holds.

**Pattern matching**

The patterns associated with a domain are known as object template expressions. A template expression match results in a binding of elements from the typed model to variables declared in the domain. A template expression match may be performed in the context that some of its variables have been bound to model elements (from the evaluation of the guard clause or other template expressions). In this case, the template expression match only finds bindings for the free variables of the domain.

In the example, for the domain associated with the `uml` model, the pattern binds all declared variables `c`, `p` and `cn`, starting from the root variable `c` of type `Class`. However, because the guard clause has been evaluated, the variable `p` has been bound to some `Package` element of the `uml` model. The pattern then finds all `Class` elements satisfying the constraints that its property `kind` having a literal value of `'persistent'` and it belongs to the namespace by the `Package` element `p`. A comparison between a property and a free variable like `name=cn` would bind the variable to the property `name` of the bound class `c`; the binding of this variable will be used in another domain or in the evaluation of the where clause.

The matching continues with nested template expression. In the example, an empty nested template expression `namespace=p:Package{}` is provided, but any other template expression may be given here to constrain the binding of elements to variables. Arbitrary deep nesting of template expressions is permitted, and matching and variable binding proceeds recursively until there is a set of value tuples corresponding to the variables of the domain and its template expressions.

**Key and object creation**

The Relations language allows determining whether a new object should be created in the target model by defining keys for different types of elements. A key determines uniquely the object in the model. For example, `Table` in a `SimpleRDBMS` model has a key based on identifying properties `schema` and `name`. The object template expression in the domain of the `rdbms` model uses properties appearing in this key, and thus the key is used to locate the table in the `rdbms` model; a new table is only created when a matching table does not exist.

**Graphical syntax**

The specification also proposes a graphical syntax to define transformation rules using graph. Figure 15 shows the corresponding graphical definition of the transformation relation `ClassToTable` in the above example. Please refer to the specification for detailed introduction of graphical notation elements [23].

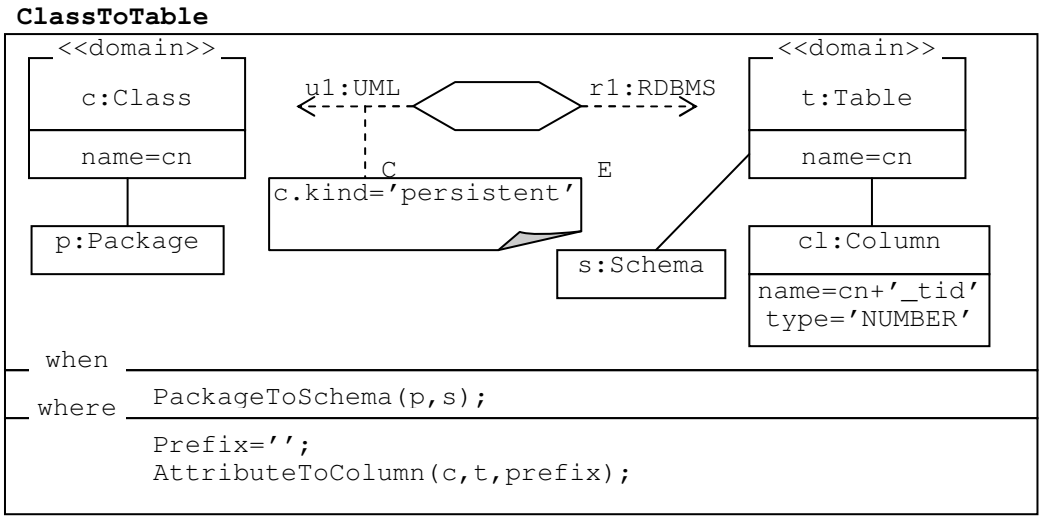



***Figure 15: Class to Table in graphical syntax [23]***

### 2.4.3 The Core Language

As introduced, the Core language of the QVT specification is used to define the transformation rules at a different level of abstraction which is simpler, but more verbose. The main difference is that trace classes are not derived implicitly from the relations as in the Relations language, but are created explicitly as other classes. Table 2 shows the Core transformation definition corresponding to the Relations example in Table 1, while Figure 16 presents the area-pattern representation of one of the transformation rules of this transformation definition. This section provides the explanation of the basic concepts used in the Core language using this example.

| Area for Domain `uml` | Middle Area | Area for Domain `rdbms` | |
|---|---|---|---|
| `p:Package` | `p2s:TPackageToSchema`<br>*`p2s.p=Package`*<br>*`p2s.s=Schema`* | `s:Schema` | Guard patterns |
| `c:Class`<br>*`c.kind='persistent'`* | **`c2t:TClassToTable`**<br>`cn:String`<br>`prefix:String`<br>*`c2t.p:=p, c2t.c:=c,`*<br>*`c2t.s:=s, c2t.t:=t,`*<br>*`cn:=c.name,`*<br>*`t.name:=cn`* | **`t:Table`**<br>*`t.schema:=s`* | Bottom patterns |

Notes: `Normal font` : variables
**`Bold font`** : realized variables
*`Italic font`* (= operator) : constraints
*`Italic font`* (:= operator) : derivations

***Figure 16: Area-pattern representation of the ClassToTable mapping***

**Transformations and directions**

A *transformation* is defined in a package and has multiple *directions* which define the candidate models of the transformation. Each direction may import one or more packages which define the allowable types of elements of the transformed models. At execution time, one of the directions are specified as the target, while the others are the sources.

The example defines the `umlRdbms` transformation in a module and the transformation includes two directions named `uml` and `rdbms` which import `SimpleUML` and `SimpleRDBMS` packages, respectively.

**Mappings, domains and areas**

*Mappings* are similar to relations in the Relations language and are part of the transformation. Each mapping has zero or more *domains*; each domain is associated with one *direction* of the transformation, thus it represents the candidate model involved in the transformation. A *domain* may be marked as checked or enforced (which also means checked) or nothing at all. The example shows two domains having a same name with their directions `uml` and `rdbms`; the former is checked, while the latter is enforced.

Mappings are divided into *areas*. The `classToTable` mapping has three areas; one area for each domain (represented by the same name of the domain; i.e. `uml` and `rdbms`) and one middle area (represented by the `where` clause). Each area in turns consists of two *patterns*: the *guard pattern* (the part in the parenthesis before the curly braces; e.g. '`p: Package`' in the `uml` area) and the *bottom pattern* (the part inside the curly braces). See below for explanation of patterns, and guard and bottom patterns. Figure 16 shows the division of this mapping.

```
module UmlRdbmsTransformation imports SimpleUML, SimpleRDBMS {
    transformation umlRdbms {
        uml imports SimpleUML;
        rdbms imports SimpleRDBMS;
    }
    class TClassToTable {
        p: Package;
        c: Class;
        s: Schema;
        t: Table;
        cl: Column;
        k: Key;
    }
    map classToTable in umlRdbms {
        check uml (p: Package) {
            c: Class |
            c.kind = 'persistent';
        }
        check enforce rdbms (s: Schema) {
            realize t: Table |
            t.schema := s;
        }
        where (p2s: TPackageToSchema | p2s.p=p; p2s.s=s) {
            realize c2t: TClassToTable, cn: String, prefix: String |
            c2t.p := p; c2t.c := c;
            c2t.s := s; c2t.t := t;
            prefix := '';
            cn := c.name;
            t.name := cn;
        }
        map {
            check enforce rdbms () {
                realize cl: Column |
                t.column := cl;
            }
            where () {
                c2t.cl := cl;
                cl.name := cn + '_tid';
            }
            map () {
                where () {
                    cl.type := 'NUMBER';
                }
            }
        }
        map {
            check enforce rdbms () {
                realize k: Key |
                k.column := cl;
                t.key := k;
            }
            where () {
                c2t.k := k;
                k.name := cn + '_pk';
            }
        }
    }
}
```

***Table 2: UML to Relational transformation definition in Core (partly) [23]***

**Patterns**

A pattern is a set of *variables* and *constraints* or *derivations*. In the example in Figure 16, the guard pattern (the upper one) of the `uml` domain contains a single variable `p` of type `Package`, while the bottom pattern (the lower) consists of another variable `c` of type `Class` and a constraint that its `kind` property should have value '`persistent`'. The types `Package` and `Class` must be defined or imported from the `SimpleUML` package that is imported by the `uml` direction of the `uml` domain. The package of the transformation defines or imports (implicitly from the QVT language) the types of variables of the middle patterns, such as the defined tracing class `TClassToTable`, or the primitive type `String`.

During execution, variables of a domain area are bound to elements of the corresponding model associated with that domain, while values of variables of the middle area are calculated based on domain variables. For example, variable `c` of the `uml` domain is bound to classes in the `uml` model, while variable `cn` of the middle is assigned with the `name` property of the class bound to variable `c`.

Variables may be *realized*; i.e. their values are created when there is no existing value satisfying the relationship. This is usually the case with variables of an enforced domain, such as the variable `t` of type `Table` of the `rdbms` domain, or a special variable of bottom pattern of the middle area representing the tracing instance of the mapping, like the variable `c2t` of type `TClassToTable` (see Figure 16).

Patterns also contain *constraints* or *derivations*. Constraints are checked to see whether a value is a valid binding for a variable, while derivations are enforced, i.e. values are created and bound to variables if no valid value exists. Thus derivations only appear in the bottom pattern of an enforced domain and they are only effective when the model of that domain is the target of the execution of that transformation. When it is used as a source, its derivations are changed to constraints. The operator symbols of constraints and derivations are '=' and '**:**=', respectively. See Figure 16 for an example.

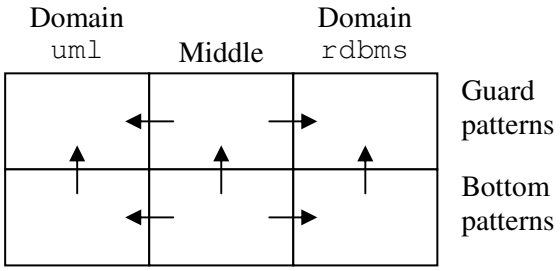



***Figure 17: Dependencies between patterns [23]***

Patterns may depend on each other; i.e. they use variables of other patterns. For example, in Figure 16, the bottom pattern of the middle area uses variable `c` of the bottom pattern of the domain area `uml`. Some common rules of dependency are shown in Figure 17. A middle pattern depends on all domain patterns at the same level (either guard or bottom). The bottom pattern depends on the guard pattern of the same area (either domain or middle area). Finally, when a direction A uses another direction B, then each pattern of the domain dA which has direction A depends on the domain dB which has direction B at the same level (either guard or bottom).

**Bindings**

A *binding* is a unique set of values for all variables of a pattern. A *valid binding* is a binding in which all variables are bound to a value other than `undefined`, and all constraints are evaluated to `true`. A *partial binding* is a binding where one or more variables are bound to a value other than `undefined` or one or more constraints evaluate to `true`, no constraints evaluate to `false` (thus, either `true` or `undefined`). Finally, an *invalid binding* has at least one constraint which evaluates to `false`.

Bindings may also depend on each other. If a pattern C depends on patterns S1…Sn, then a valid binding of C needs one valid binding for each S1…Sn. A valid combination of valid bindings VSB of a set of dependent patterns SP is a set of valid bindings in which each pattern of SP has one unique valid binding from VSB, and each valid binding of VSB uses other valid bindings of VSB according to the dependencies between patterns of SP.

**Guard and bottom patterns**

With the definition of binding, the difference between guard and bottom patterns can be clarified. The matching of all bottom patterns takes place in the context of a valid combination of valid bindings of all guard patterns. The validity of the combination is defined by the dependencies between the guard patterns.

Note that the guard patterns do not define any constraint or derivation on the transformed models. They only define the context in which the constraints and derivations in the bottom patterns are evaluated or calculated. Thus the guard patterns narrow the choice of values in models for just one mapping, but not for the model as a whole.

For example, the guard patterns in the example limit the derivation of a RDBMS table from a UML class in a context that (1) the variable `p` of type `Package` (defined in the guard pattern of the domain `uml`) is bound to some package of the `uml` model, and (2) the variable `s` of type `Schema` (defined in the guard pattern of the domain `rdbms`) is bound to some schema of the `rdbms` model, and (3) these package and schema elements appears in one instance of the class `TPackageToSchema` (defined in the middle guard pattern); i.e. the schema was derived from the package in a previous mapping execution.

**Mapping composition**

Mapping composition allows defining mappings that are only "active" in a context given by another mapping. The child mapping is executed in the context of a valid binding of the bottom-middle pattern of the parent mapping. This mechanism is shown in the example where a column is derived in a child mapping in the context of a valid binding of the `TClassToTable c2t` variable of the bottom-middle pattern of the parent mapping. This is necessary to avoid the problem of unwanted creation or deletion of elements in the target model by first checking and deriving the table, then checking and driving the column which is not an identifying property of the table.

**Tracing**

The QVT languages in this QVT specification use trace classes for keeping traces between source and target elements which are mapped in a transformation. However, in the Relations language, these trace classes are not explicitly specified and used. A relation directly specifies the relationship between source and target domain elements. On the other hand, in the Core language, a trace class is specified explicitly for each transformation mapping. The example shows the specification of the trace class `TClassToTable` corresponding to the mapping `ClassToTable`.

The QVT specification also proposes transformation rules to transform a transformation definition in the Relations language to the corresponding one in the Core language, in which there is a rule to define the a trace class for each relation. The rule is described as follows:

*"Corresponding to each relation there exists a trace class in core. The trace class contains a property corresponding to each object node in the pattern of each domain of the relation."* [23]

## 2.5 Summary

This chapter presented the basic concepts in the Model Driven Architecture. MDA is a software development approach which bases on models and model transformations. The main concepts of the approach are models, metamodels and transformations. The Model Driven Engineering is the generalized approach of MDA. It is considered that MDA is an instance of MDE implemented on a set of technologies proposed by the Object Management Group.

In MDA, models are the primary artifacts, and model transformations play a key role. Model transformations in MDA are based on metamodels. They specify the mapping between source and target elements whose types are defined in the source and target metamodels. Upon execution, these mappings are used to derive elements in the target model from corresponding elements in the source models.

The QVT language proposed by the QVT Merge Group is the convergence of various QVT submissions in response to the Request for Proposal for MOF 2.0 QVT by OMG. This QVT language is considered to be standardized in the near future, and it will be used in the case study of this research.

The QVT language is a hybrid of declarative/imperative approach, with the declarative part being split into two sub-languages: Relations and Core. Both languages allow a declarative specification of transformation rules as relationships between MOF models. However, trace classes and their instances which are used to record the mapping between source and target elements occurred during a transformation execution must be explicitly defined and created in the Core language, whereas they are deduced implicitly from the transformation definition in the Relations language.

Finally, the Core language is considered to be more formal and could be used as a reference of implementation for the Relations language. The explicit division of a transformation rule written in the Core language into areas and patterns is the base for identifying the mapping between types of elements of models involved in that rule. It also differentiates the roles of these elements in that mapping. This information will be used to derive dependency graphs of transformation rules at metamodel level in the case study.

# 3 Crosscutting Concerns in AOSD

This chapter discusses crosscutting concerns and related concepts in Aspect-Oriented Software Development (AOSD). The problem of crosscutting concerns is introduced in Section 3.1. Section 3.2 introduces Aspect-Oriented Programming (AOP) as a new programming paradigm to cope with crosscutting concerns at the programming level. The related concepts such as scattering, tangling and crosscutting are then discussed by providing several frameworks for defining these concepts in Section 3.3. Finally, Section 3.4 gives the rationale for selecting a particular framework to use in this research and summarizes the chapter.

## 3.1 Problem of Crosscutting Concerns

Separation of concerns is a key principle in software engineering [8][30]. The principle states that a given problem involves different kinds of concerns, which should be identified and modularized to cope with complexity and to achieve quality factors such as robustness, adaptability, maintainability and reusability.

According to [30], a concern is "*a canonical solution abstraction that is relevant for a given problem*". This definition implies that the solution for the given problem should meet two quality properties. Firstly, the solution should be relevant; i.e. it is required to meet the goals and constraints of the problem, and as such is valid for the context of the problem. Secondly, the solution should include necessary and sufficient abstractions to provide a relevant solution. Furthermore, it should not include irrelevant and/or redundant abstractions. In other words, the solution should be generic and succinct, that is, canonical.

The decomposition of a problem into multiple concerns may occur in many different ways. A concern may occur in a composition, but may not in another composition because it is not relevant to the stakeholders. Furthermore, the result also depends on the used view and methodology. For example, an object-oriented methodology may model concerns as classes and objects which are derived from requirements, whereas these concerns are represented as procedures in a procedural language.

The separation of concerns principle states that each concern of the given problem should be mapped to one module in the system. In other words, the problem should be decomposed into modules such that each module has one concern. By this way, the concerns are localized and then can be easier to be understood, extended, adapted, reused, etc. This process is illustrated in Figure 18. In this figure, the problem is decomposed into concerns $C_1$ to $C_n$ and each of these concerns is implemented in a separate module M. A module is a modular unit in the given design language (class, function, procedure, etc.)

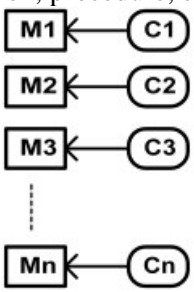



***Figure 18: Mapping Concerns $C_1..C_n$ to Modules $M_1..M_n$ [30]***

Unfortunately, there are concerns that cannot be implemented in a single module, such as security and logging. These concerns are said to crosscut the boundaries of the implementing modules. Even concerns which are well separated at the design level become crosscutting at the programming level. A crosscutting concern is a serious problem because it makes the program harder to understand, reuse, extend or adapt, as it spreads over many places. Finding where a crosscutting concern occurs is one problem, while adapting the concern appropriately is another problem. Figure 19 illustrates the problem of crosscutting concerns. In this figure, the concern C3 is mapped to multiple modules M2, M3, M4 and M5. The problem is even

worse when there are multiple crosscutting concerns. Because of these crosscutting concerns, several modules may include more than one concern. These concerns are said to be tangled in the corresponding module. For example, the concerns C2 and C3 are tangled in the module M2, even though the C2 is not a crosscutting concern.

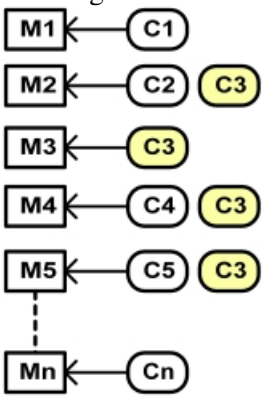



*Figure 19: Concern C3 crosscuts modules M2, M3, M4 and M5 [30]*

[30] presents the crosscutting and tangling problem in another way. In Figure 20, the modules form the vertical axis, while the concerns locate over the horizontal axis. Each circle in the figure represents a place where a concern crosscuts a module. These are called joinpoints. By looking at this figure, we can easily identify which concerns are crosscutting which modules, and which modules become tangling.

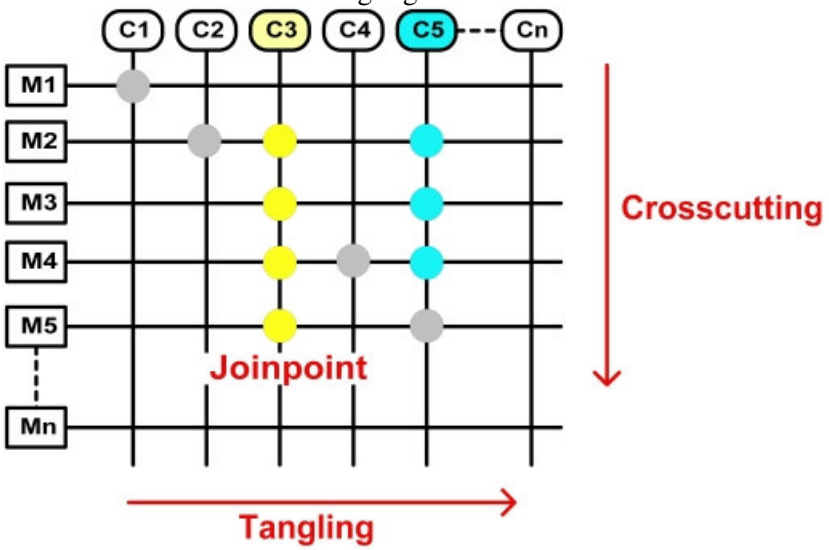



*Figure 20: Crosscutting, Tangling and Joinpoints [30]*

In order to solve the problem of crosscutting concerns, the Aspect-Oriented Software Development (AOSD) approach needs to be applied. In the remaining sections, we discuss Aspect-Oriented Programming as an AOSD approach at the programming level and present several frameworks for defining the related concepts.

## 3.2 Aspect-Oriented Programming

Several aspect-oriented programming languages have recently emerged to solve the crosscutting concern problem at the programming level. Some of them are AspectJ, Composition Filters, and HyperJ. AspectJ [1][8] is a general purpose extension to Java that introduces new language constructs to represent and compose aspects. HyperJ [10] supports multi-dimensional separation of concerns for Java and operates on standard Java class files and produces new class files for execution. In Composition Filters [5][8][28], aspects are represented through the so-called composition filters that are declaratively specified and integrated in the corresponding programming language. In this section, we introduce the general concepts used in these AOP languages.

**Aspects**

In these AOP languages, crosscutting concerns are identified and implemented in a separate module which is called an aspect. In these aspects, the extension to other modules of the system, including the location and behavior, is defined. These extensions are later woven to the extended modules through an aspect weaving process.

**Joinpoints**

A joinpoint is a well-defined point in the program flow where the extension defined by an aspect needs to be applied. A pointcut picks out certain joinpoints and values at those points. Some kinds of joinpoints are given in Table 3 below.

| Kind of joinpoint | Description |
|---|---|
| Method call | When a method is executed |
| Method execution | When the body of a method is actually executed |
| Exception handler execution | When an exception is thrown |
| Field access | When a non-constant field is referenced or set |

***Table 3: Several kinds of joinpoints [1][8]***

**Advice**

Advice is used to define the behavior at the crosscutting points, or joinpoints. Advice brings together a pointcut (to pick out joinpoints) and a body of code (to be executed at each joinpoint). In AspectJ, an advice can be specified to run before, after or around a joinpoint. Before-advice runs as a joinpoint is reached, before the program proceeds with the joinpoint. After-advice on a particular joinpoint runs after the program proceeds with that joinpoint. Around-advice on a joinpoint runs as the joinpoint is reached and has explicit control over whether the program proceeds with the joinpoint or not.

Composition Filters controls the execution of an advice through a set of filter specifications. A filter has a certain type and some of them are listed in Table 4 below. A filter can be guarded by a condition and has an associated set of pointcuts.

| Filter type | Accept action | Reject action |
|---|---|---|
| Dispatch | Dispatch the message to the target of the message | Continue to the next filter, or raise an exception if there is no next filter. |
| Error | Continue to the next filter, or raise an exception if there is no next filter | Raise an exception |
| Wait | Continue to the next filter, or raise an exception if there is no next filter | The message is placed in a queue as long as the evaluation of the filter results in a rejection |
| Meta | The reified message is sent as a parameter of another message to an object. The receiving object can observe and manipulate the message, then reactivate the execution | Continue to the next filter, or raise an exception if there is no next filter |

***Table 4: Several kinds of filters in Composition Filters [5]***

## 3.3 Definitions of Crosscutting and Related Concepts

As discussed in the previous sections, several aspect-oriented programming languages and tools have emerged to solve the problem of separation of concerns. Specifically, these languages introduce new elements to enhance the expressiveness in order to cope with scattering, tangling and crosscutting issues. However, these languages do not provide precise definitions for the concepts of scattering, tangling and crosscutting. They all provides necessary elements to declare aspects, joinpoints and advices as introduced in the previous section, but they use different terminology and their definitions are different.

This variation raises a need to have a formal framework to understand and compare how these aspect-oriented languages support the modular crosscutting. Furthermore, when the aspect-oriented technique is brought to other phases of the software development process such as designing, requirements engineering, there is a need to have a consensus definition for these related concepts. Several researches have recently proposed frameworks in which formal definitions for the concepts of scattering, tangling and crosscutting are given [3][13]. This section introduces some of the frameworks proposed by these researches and discusses how these frameworks can be applied to the MDA framework and QVT model transformations.

### 3.3.1 Modeling Crosscutting in Aspect-Oriented Mechanisms

In aspect-oriented languages, the problem of separation of concerns is solved by allowing the developers to declare crosscutting concerns in separate modules and a set of joinpoints where these modules crosscut each other. At compilation time and/or runtime, a weaving process takes these modules and weaves them into a single combined module/computation. Masuhara and Kiczales captured the core semantics of the aspect-oriented mechanism by modeling this weaving process [13]. Each weaver – the term for the corresponding AO mechanism – is modeled by an 11-tuple :

$$<X, X_{jp}, A, A_{ID}, A_{EFF}, A_{MOD}, B, B_{ID}, B_{EFF}, B_{MOD}, META>$$

In this tuple:

- A and B are the languages of the input programs, while X is the result domain of the weaving process which may either be a computation or a third language.
- $X_{jp}$ is the join points in X, whereas $A_{ID}$ and $B_{ID}$ are the means, in the languages A and B, of identifying elements of $X_{jp}$.
- $A_{EFF}$ and $B_{EFF}$ are the means, in the languages A and B, of effecting semantics at the identified join points. $A_{MOD}$ and $B_{MOD}$ are units of modularity in A and B.
- META is the optional meta-language for parameterizing the weaving process.

Table 5 provides examples for elements of this weaver model in the AspectJ language.

| | **AspectJ** |
|---|---|
| X | The program execution |
| $X_{jp}$ | Method calls, method execution, field accesses, etc. in the program execution where an advice is weaved. |
| A | Declarations of classes, methods, fields of the Java language |
| $A_{ID}$ | Method, field signatures |
| $A_{EFF}$ | Execution of the method body, access the field |
| $A_{MOD}$ | Class, package |
| B | Declarations of advices (AspectJ extension to the Java language) |
| $B_{ID}$ | Pointcuts |
| $B_{EFF}$ | Execution of the advice body |
| $B_{MOD}$ | Advice |
| META | None |

***Table 5: Examples of elements of the weaver model [13]***

The weaving process is defined by a procedure with the following signature:

$$A \times B \times META \rightarrow X$$

One of the critical properties of this model is that the join points are considered to be in the combined result, but not in the input programs. The model also describes A, B and X as distinct entities and describes the weaving process as the combination of semantics of the input programs of A and B at the join points in X. This perception is different from other models which consist of only two elements (A and B) and consider a program in B to be weaved with a program in A at join points in A.

Before defining how two modules of the input programs in A and B crosscut each other, the authors defined the projection of a module, let's say a module $m_A$ from the input program

$p_A$ in A, onto X as the set of join points identified by the elements $A_{ID}$ within $m_A$. For example, in AspectJ, the projection of an advice declaration is the set of join points matched by the pointcut of that advice declaration.

Finally, the authors defined the concept of crosscutting: "*For a pair of modules $m_A$ and $m_B$ (from $p_A$ and $p_B$), mA is said to crosscut mB with respect to X if and only if their projections onto X intersect, and neither of the projections is a subset of the other*". Figure 21 illustrates this definition. In this figure, the dashed boxes in A and B represents the modules $m_A$ and $m_B$, respectively. The dots in these boxes are the identification of join points $A_{ID}$ and $B_{ID}$ in these modules. The lines between these dots and the corresponding ones in X show the projection of the modules onto X. These projections intersect and neither of them is a subset of the other, so the modules in A and B crosscut each other.

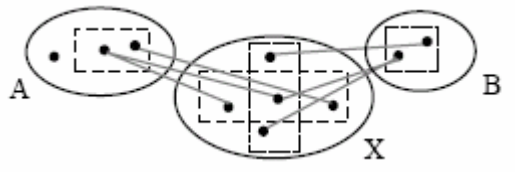



***Figure 21: Modular crosscutting [13]***

This framework has successfully captured the mechanisms of AO programming languages and visualized how the modules from the input programs crosscut each other with respect to the combined computation/program. However, the framework seems focus too much on the programming level. The paper provides discussion and examples on several AO programming languages but not on the general context of software development. Finally, the framework does not define other concepts like scattering and tangling which are important properties of crosscutting.

### 3.3.2 Disentangling Crosscutting in AOSD

Berg and Conejero [3] fill in this gap by providing another conceptual framework in which the concepts of scattering, tangling and crosscutting are defined explicitly. The description of crosscutting in this conceptual framework is similar to some descriptions in the framework proposed by Masuhara and Kiczales [13].

In this framework, scattering, tangling and crosscutting are defined in terms of "one thing" with respect to "another thing": there are two domains, or two levels, or two phrases, or two layers which relate to each other in some way. Examples for these two things are two domains in mathematical sense in which there is a mapping from one domain into another domain, or two phases in the software development cycle, or two levels in the Model Driven Architecture. They are represented by the terms source and target.

The relationship between the source and the target is represented by the Crosscutting Pattern shown in Figure 22. In the Crosscutting Pattern, elements in the source are related to elements in the target. This relationship is symmetric. The terms scattering, tangling and crosscutting are defined as special cases of this relationship. This is explained by extending the Crosscutting Pattern with the Mapping concepts, as shown in Figure 23.

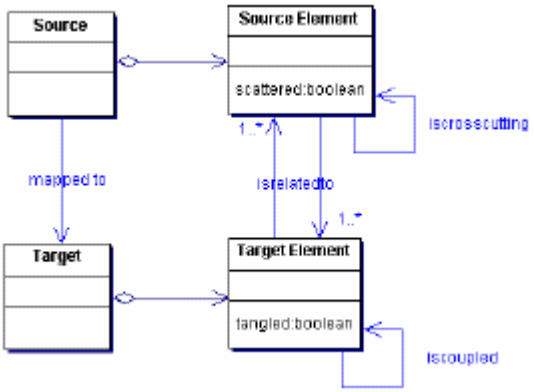



***Figure 22: Concept Diagram of Crosscutting Pattern (without Mapping Concepts) [3]***

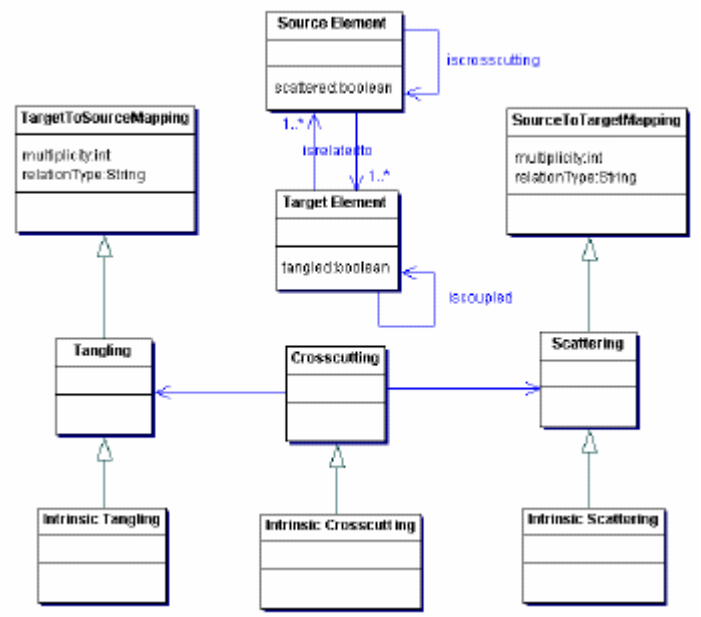



***Figure 23: Concept Diagram of Crosscutting Pattern (with Mapping Concepts) [3]***

In Figure 23, `SourceToTargetMapping` is the relation between the source elements and the target elements. The multiplicity of the relation can be 1:1 or 1:many. Scattering is defined as follows: "*Scattering occurs when, in a mapping between source and target, a source element is related to multiple target elements*." In this case, the source element is said to be scattered over target elements.

The reverse of the above mapping is `TargetToSourceMapping` which is the relation between the target elements and the source elements. Similarly, the multiplicity can be 1:1 or 1:many, and tangling is defined as follows: "*Tangling occurs when, in a mapping between target and source, a target element is related to multiple source elements*." Another way of statement is: *two source elements are tangled if they are mapped onto the same target element*.

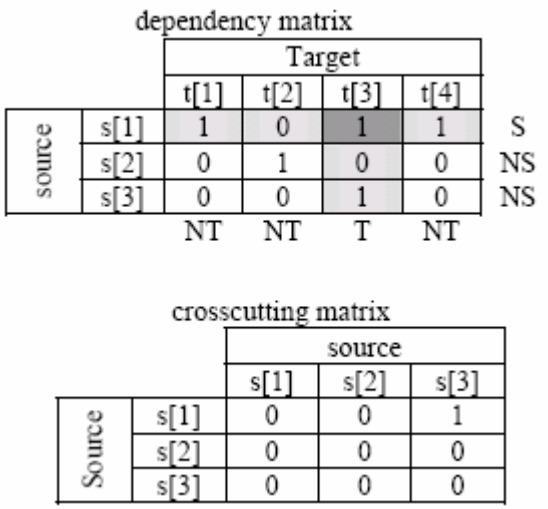
A specific combination of scattering and tangling results in crosscutting which is defined as follows: "*Crosscutting occurs when, in a mapping between source and target, a source element is scattered over target elements and where in at least one of these target elements, some other source elements are tangled.*"

In this crosscutting, a source element s1 is said to crosscut another source element s2 if s1 is scattered over target elements and s1 is tangled with s2 over at least one of these target elements. However, the source element s2 is not required to be scattered. That means this definition is not symmetric and less restrictive than the crosscutting definition of Musuhara and Kiczales [13].

The authors also provide a technique to determine scattering, tangling and crosscutting based on dependency matrix and crosscutting matrix. A *dependency matrix* (source x target) represents the dependency relation between source elements and target elements. The source elements are in the rows and the target elements are in the columns. A cell of 1 denotes the relation between the corresponding source and target elements. Scattering and tangling are easily visualized in this matrix, as shown on the example in Figure 24.

A new auxiliary concept is defined for the *crosscutpoint* in the dependency matrix: it is any cell which involves in both scattering and tangling. It is said that there is crosscutting if one or more crosscutpoints exist in the dependency matrix.

A *crosscutting matrix* (source x source) represents the crosscutting relation between source elements, for a given source to target mapping (represented in a dependency matrix). A cell of value 1 denotes that the source element in the row crosscuts the source element in the column. A technique is also provided to derive the crosscutting matrix from the given dependency matrix.




*S: scattered source element* *T: tangled target element*
*NS: non-scattered source element* *NT: non-tangled target element*

***Figure 24: Dependency and crosscutting matrices with tangling, scattering and crosscutting [3]***

Finally, the authors define the terms *intrinsic scattering*, *intrinsic tangling* and *intrinsic crosscutting* as the case when scattering, tangling and crosscutting are caused by the limitations in the expressive power of the languages used to describe the source and target.

## 3.4 Summary

This chapter introduced the problem of crosscutting concerns in which concerns, defined as canonical solution abstractions that are relevant for a given problem, cannot be implemented in a single module, such as security and logging. These crosscutting concerns make the program harder to understand, reuse, extend or adapt, as they spread over many places.

Aspect-Oriented Software Development provides techniques to cope with the problem of crosscutting concerns at various phases of the software lifecycle. Aspect-Orient Programming is the corresponding technology at the programming level. There are several AOP programming languages such as AspectJ, Composition Filters and HyperJ; they introduce new language constructs to represent and compose aspects – a kind of modules to implement crosscutting concerns.

The final section of the chapter presented two frameworks for defining the concepts of scattering, tangling and crosscutting. At the first glance, these two frameworks seem to propose inconsistent and conflict definitions. However, a closer look at them shows that they use the same approach to define crosscutting in terms of "one thing" with respect to "another thing".

In Berg and Conejero's framework, a source is mapped to a target by a set of relations between source and target elements, as shown in Figure 22 and Figure 23. Masuhara and Kiczales propose the same mapping approach, but with a slight difference that the source is split into two input programs, as shown in Figure 21. Both frameworks then define crosscutting as a specific situation of this mapping.

However, Berg and Conejero's framework has several advances over Masuhara and Kizales'. First, the latter focuses too much on the programming level, while the former discusses the problem in the general context of software development. It uses the term source and target to represent arbitrary things, such as two domains, two levels, two phases or two layers which relate to each other in some way. In MDA, they could be two models related by model transformations.

Next, the latter only proposes a pattern of mapping between input programs and the result computation/program, but does not specify explicitly how to identify the relations between them, and only crosscutting could be determined. The former extends the pattern with

relations between source and target elements, and provides a specific technique to determine scattering, tangling and crosscutting based on dependency graphs, dependency matrices and crosscutting matrices.

The definition of concepts of scattering, tangling and crosscutting of Berg and Conejero seems also conflict with the general description of the problem of crosscutting concerns in Section 3.1. However, they again have the same approach, but use a different terminology. In Figure 20, if we consider the concerns $C_1$,..., $C_n$ as elements of the source and $M_1$,..., $M_n$ as elements of the target, then this figure looks similar to the dependency matrix in Figure 24. In those figures, the term 'tangling' has the same meaning, while the term 'crosscutting' in the former has the same meaning as the term 'scattering' in the latter. Hence the later could be used as the formal definition for the former with a notice of this terminology difference.

Because of these reasons, the framework proposed by Berg and Conejero could be used broadly in software development. It is also very well appropriate to this research with the application of MDA and QVT model transformations. In MDA, the software application is modeled as models at various abstract levels. QVT model transformations are used to transform source models to target models. These transformations are actually a mapping between elements in the source and target models. A QVT transformation language like the one proposed by the QVT Merge Group as discussed in Section 2.3 specifies explicitly the traceability of this mapping. This traceability directly forms the dependency matrix and the crosscutting matrix based on which the different cases of scattering, tangling and crosscutting are easily identified.

# 4 Case: Concurrent File Versioning System

This chapter presents a case study extracted from the Master thesis of Henninger [9]. The case study is the development of a file versioning system which is based on "Concurrent Versions Systems (CVS)" with simplification. It will be used for our discussion about the impact of crosscutting concerns on QVT model transformations and is introduced in Section 4.1. Definition of transformations from the PIM model to the PSM models using the QVT language introduced in the previous section is then presented in Section 4.2. Secion 4.3 discusses tools support and Section 4.4 summarizes the chapter.

## 4.1 Simple Concurrent File Versioning System

A versioning system allows the user to keep multiple versions of data ordered by a timestamp. A user may get an existing version of data from the system, modify that data and save it back to the system as a new version without discarding the previous version of that data. Each version may either be a complete copy of the modified data or a log of changes to the previous version of data. [9]

A versioning system can be classified into several categories depending on the type and the storage mechanism of the maintained data. The data may be either system data or user data. Examples of system data are in database systems in which users concurrently access multiple copies of the same piece of data called snapshots; this maintenance is transparent to the user.

On the other hand, the user may want to maintain multiple versions of the user data. For user data maintenance, the data may be stored as records in a database system or as files and directories in a file system. A database system allows the user to keep multiple records for the same piece of data; one record for each version. Versions in this type of system are usually complete copies of user data. [9]

In file versioning systems, user data is stored as files and organized in a directory hierarchy. Each file has multiple copies and each copy is tagged with a version number and a timestamp. The case study in this assignment will only consider this kind of versioning systems.

Maintaining versions of a file can be implemented using one of the following options: [9]

- *Store the complete content of a file for each version.* This option is the fastest to retrieve any version of a file, but it consumes the most space and is inefficient to show the differences between two versions of a file. The next two options take less space and are straightforward to compare the contents of any two versions.
- *Store the changes in a forward order.* The system only keeps the complete content of the first version of a file. Each subsequent version is stored as a log of changes with respect to the previous version. This option takes the most time to retrieve the newest version of the file because the system has to apply changes to the complete content of the first version. It is simple for the system to remove the last version of the file by discarding the corresponding log of changes, whereas it has to calculate and store the complete content of the second version when the first version is removed.
- *Store the changes in a backward order.* This option is similar to the second option, but the complete content of the last version of a file and a log of changes for preceding versions are stored instead. Retrieving the newest version is as fast as the first option, but it takes more time for the preceding versions. Removing the oldest version of the file is simple, while removing the newest version requires the system to calculate and store the complete content of the second newest version of the file. Deleting an intermediate version of a file is as complex as the second option.

### 4.1.1 Basic functionalities

A fundamental file versioning system consists of the following basic functionalities: `check-in`, `check-out`, `commit`, `update`, `remove` and `difference`. Figure 25 presents the use case diagram containing basic functionalities of a versioning system.

The `commit` functionality allows the user to create a new version a file in the system with the local copy in the user workspace, while the `update` functionality does the opposite: to update the local copy in the user workspace with the content of a specific version of the file in the system. Both use cases require a conflict management; i.e. when the local copy of the file in the user workspace and the current version of the file in the system are modified simultaneously.

The `check-in` and `check-out` functionalities, besides similar operations as `commit` and `update`, manage the locking mechanism. The `check-out` functionality locks the file before retrieving the content, whereas the `check-in` functionality creates a new version of the file in the system and releases the lock on that file. The locking mechanism is necessary for concurrent access to a file in the system by multiple users.

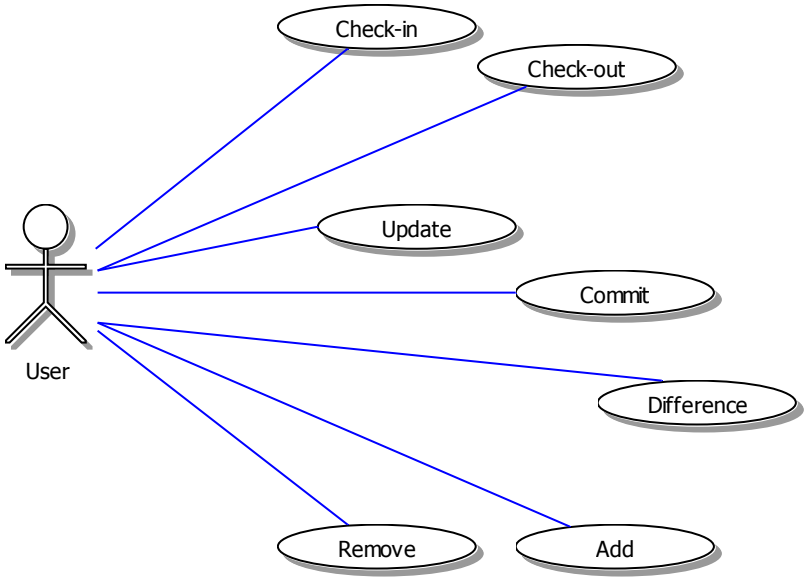



***Figure 25: Basic versioning system use case diagram [9]***

The `remove` functionality deletes the specified version of a file from the system. As noted above, the system may need to calculate the complete content of the first or last version of the file in the case of the second and third options, respectively. The `difference` functionality shows the differences between two versions of a file.

### 4.1.2 Branching and tagging

The case study becomes more interesting when the versioning system is extended for software configuration management (SCM). Essential for SCM is the tracking and maintaining the integrity of multiple files. Some features needed for this requirement are `branching` and `tagging`.

A software product usually consists of multiple files (documents, source codes, etc.); each file in turns has multiple versions which are all managed by a versioning system. A version of a software product is then a set of files; each of which has a specific version number. `Tagging` is the ability of the versioning system to mark a specific version of each of these files to belong to a specific version of the software product.

`Branching` is related to the following definitions:

- The *main trunk* is the main development line of the software product.
- A *branch* is a parallel version of the main trunk or another branch.

During the software product development, a separated version may be branched from the main version of the software product. This is usually necessary to do bug fixes or to test a new feature of the software product without affecting the main trunk. This is also the case of software product line engineering in which multiple lines of a software product are maintained. The branch may be later merged with the main trunk or another branch and conflict management is needed as with the use cases `update` and `check-in`.

With a versioning system, `branching` is the ability to branch a set of tagged files off the main trunk or another branch. A branch can only be conducted after the set of files has been tagged. `Merging` is the ability to merge a branch back to the original branch from which it was originated. Figure 26 shows the use case diagrams for these extended functionalities.

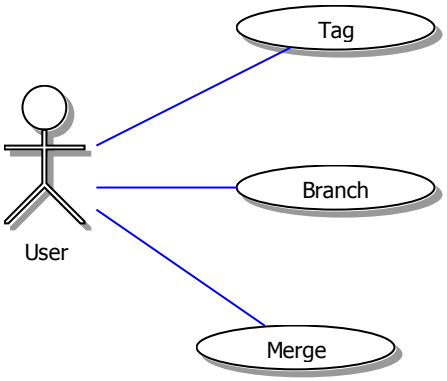



***Figure 26: Extended versioning system use case diagram [9]***

### 4.1.3 Security and Persistence

The CFVS system is concurrently used by multiple users with different permissions. This means that it has to implement the security functions to manage the permissions of each user. The two most fundamental security functions that are implemented by this system are authentication and authorization. *Authentication* is the requirement that the user must be authenticated before using the system. Each user has a username and a password which are entered at the beginning of his or her working session with the system. The system checks whether the username exists and the password matches.

*Authorization* is the ability of the system to check whether a user is allowed to do some action on a particular object. The CFVS has multiple kinds of objects such as product, branch, directory and file, each object is associated with allowable actions; the rights to do these actions are assigned to individual users by the administrator. For example, a particular product would have permissions like reading its contents, branching its branches or checking out its files. For the purpose of simplicity, this CFVS system only manages the permissions on product and branch objects; other kinds of objects like directories and files have their permission assignment inherited from their parent product and branch.

Data of the CFVS system also needs to be stored on a storage media. This requirement is fulfilled by implementing the persistence functions like retrieving data from and saving data to the storage media. In this case study, a relational database is used to store basic information of objects of the system, while a file system is used to store the actual contents of files.

These security and persistence requirements are some of the most important quality requirements which are implemented by almost all software applications. They are usually considered as crosscutting concerns according the current AOP languages. Subsequent sections will provide the realization of these concerns and the basic functions of the CFVS system at two MDA levels namely PIM and PSM. Transformation definitions written in the proposed QVT language will be provided to show how these concerns at the PIM model crosscut each other with respect to the PSM models according to the formal definitions of crosscutting defined in Section 3.3.

## 4.2 Models and Model Transformations

### 4.2.1 Approach

As discussed, QVT model transformations use metamodels to define mapping between elements of the source and target models. Figure 3 shows this transformation pattern and it is extended as the process for this case study in Figure 27. Following are the main steps of the case study:

- Build the PIM model of the CFVS application which is modeled in the UML metamodel.
- Define two transformations to Relational and Java PSM models. The transformation definitions are written in the Relations and Core languages.
- Execute the transformation definitions over the PIM model to derive the Relational and Java models, respectively.

These steps are represented as gray boxes in Figure 27. Boxes in this figure also show references to corresponding sections, figures and tables in this thesis.

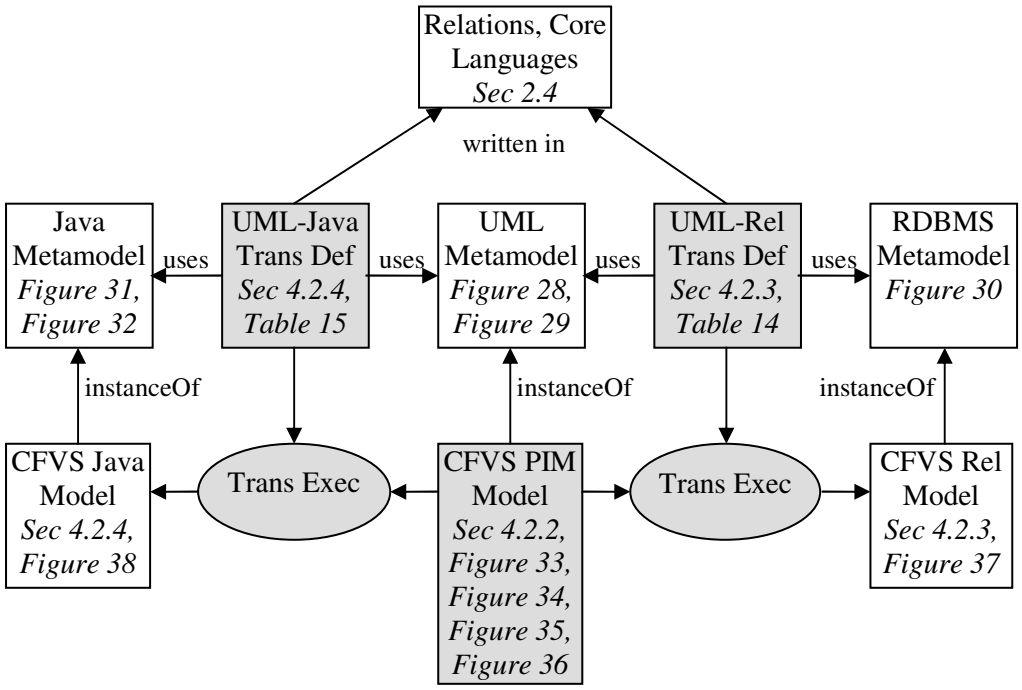



***Figure 27: Transformation process***

There are several issues in this process. The first issue is about metamodels. When transforming from the PIM model specified in the UML language to the PSM Relational model, metamodels for the UML and Relational models are needed. The ideal candidates are the UML metamodel specified the OMG's UML specification [25] and the Relational metamodel specified in the OMG's CWM specification [16]. However, these metamodels are superfluous and too complicated for most usual applications. In this case study, subsets of these metamodels are used: SimpleUML and SimpleRDBMS. These metamodels may not have been standardized, but are still able to illustrate the operational characteristic of the QVT language in discussion. The class diagrams for the SimpleUML and SimpleRDBMS metamodels are shown in Figure 28 and Figure 29, and in Figure 30, respectively.

For the Java model, the UML profile for Java specified in the OMG's Metamodel and UML Profile for Java and EJB [20] is used. This is shown in Figure 31 and Figure 32. The profile concept is a specialization mechanism defined as part of UML. A profile includes definition of a set of stereotypes, a set of related constraints and a set of tagged values. Thus a profile defines a specialized metamodel, which is a subset of the UML metamodel. In other words, a profile defines a language by reusing the UML metamodel [11].

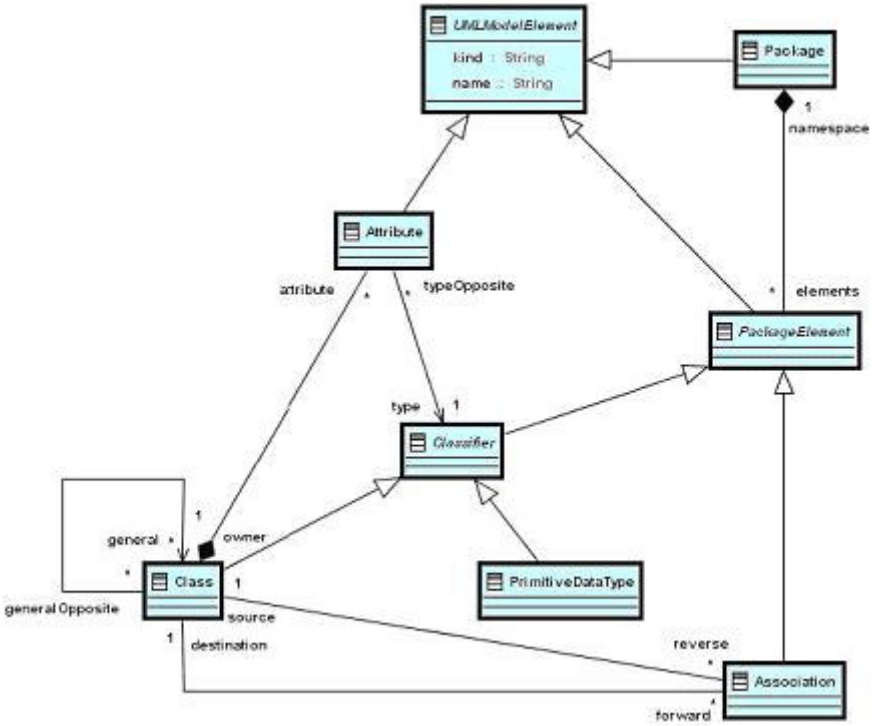



*Figure 28: Simple UML metamodel [23]*

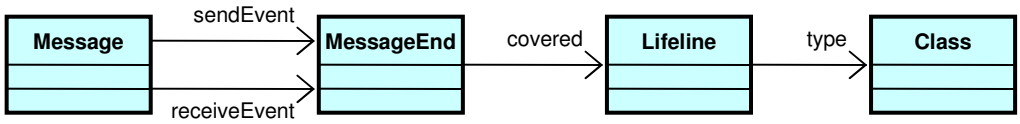



*Figure 29: The SimpleUML metamodel extended with interaction features [25]*

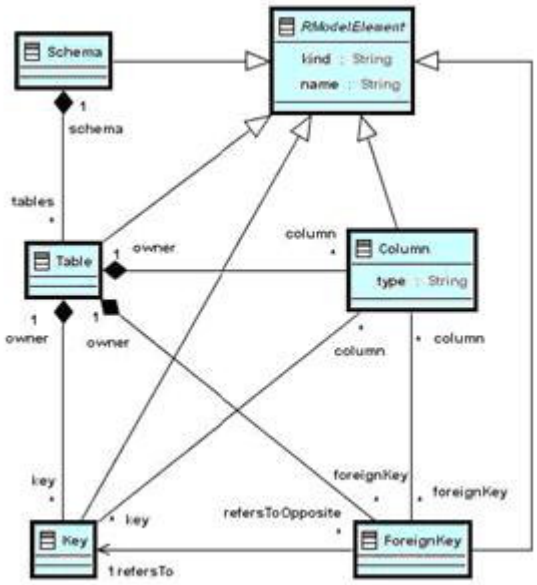



*Figure 30: Simple RDBMS metamodel [23]*

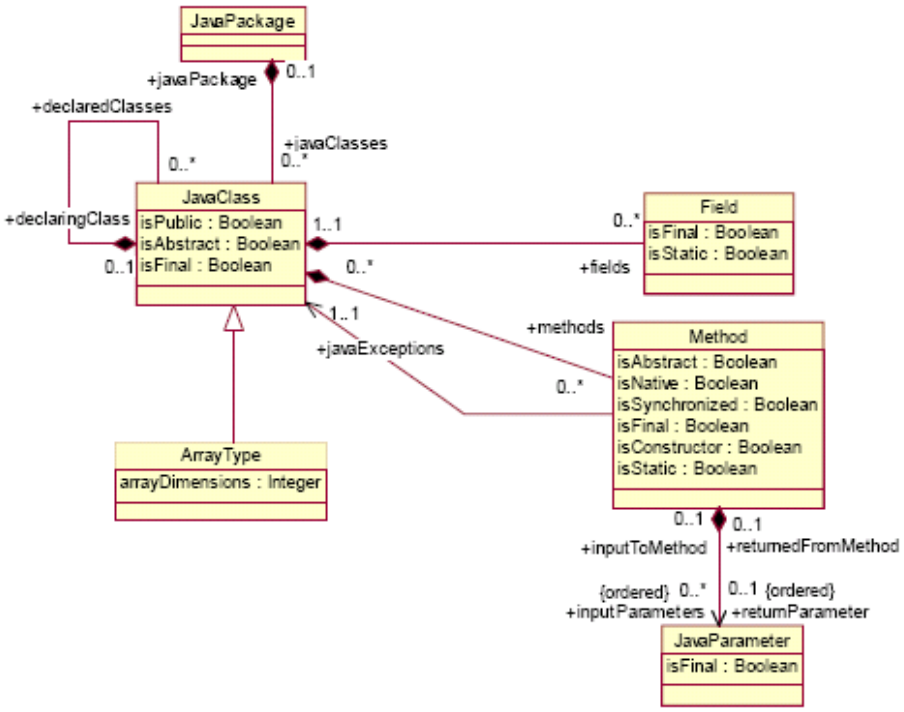



***Figure 31: The metamodel (class contents) of the UML Profile for Java [20]***

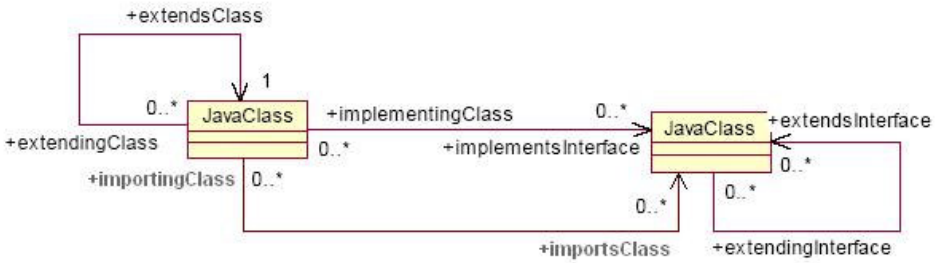



***Figure 32: The metamodel (polymorphism) of the UML Profile for Java [20]***

The second issue relates to the package information. In the UML model, classes are grouped in hierarchical packages. When transforming, these packages are usually mapped to schemas of the Relational model, or packages of Java classes. However, in the PIM model, packages may be logical grouping of classes based on, for example, different concerns or functionalities of the system. On the other hand, in the Relational model, all tables may be grouped into a single schema, as usually the case for non-distributed database applications. The same thing happens in the Java model; packages are physical grouping of Java classes which may be slightly different from logical grouping in the PIM model. The solution to this issue depends on the individual applications. One may try to keep consistent package information between different models, while the other may use a tag name (or a custom property) of classes for this purpose. In this case study, for the purpose of simplicity, the package information is not used during transformation between various models.

The third issue is that the transformation is not always from one source model to one target model, but it may be one-to-many, many-to-one or many-to-many. For example, when the layer architecture style is applied, it is usually the case that a single PIM model is designed and then transformed to multiple PSM models and eventually to multiple software components; a bridge component is also generated for communication between these components [11]. This pattern also applies to the case study in which a Relational PSM and a Java PSM model and a communication bridge in-between them are generated from a single PIM model.

### 4.2.2 Platform Independent Model

The diagrams presented in Figure 33, Figure 34, Figure 35 and Figure 36 are part of the platform independent model of the case study. It models necessary elements for constructing a simple file versioning system but does not set any restriction on a specific platform on which the system will be constructed.

Basing on the functionalities described in Section 4.1, a simple model for the versioning system can be generated. Figure 33 presents the class diagram for this simple versioning system. The class `Product` is needed for maintaining multiple software products in the versioning system. Each software product corresponds to a `Product` instance which is composed by multiple instances of `Branch`.

The initial `Branch` of a `Product` is the main trunk of the software product. It has a name and a number and it may create another branch via the operation `branch()` or be merged with another branch via the operation `merge()`.

Each branch consists of multiple files which are organized into a hierarchy of directories; these are represented by the classes `Directory` and `File`. All the basic functionalities are implemented as operations of the class `File`. They may also be implemented as `Branch`'s and `Directory`'s operations which are eventually delegated to `File`'s operations. For the sake of simplicity, this enhance is ignored in this case study.

A `File` instance holds the content of a file (either the first or the last version) and contains multiple versions which are represented by the class `Version`. Each version is stored with a timestamp and the difference with respect to the adjacent version (either the preceding or the following one, respectively).

The class `Tag` is used for tagging a set of files of a specific version. A tagged product version may contain one or more versions of files; one version for each file. A version of a file may also belong to multiple tagged product versions.

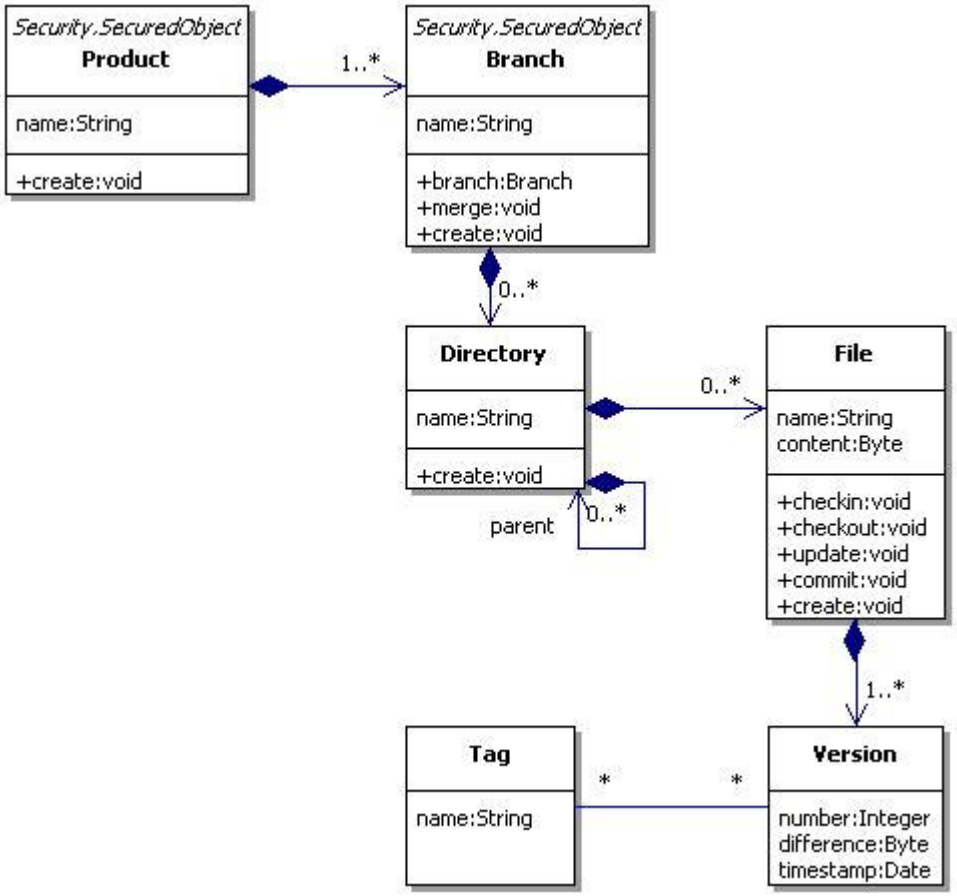



***Figure 33: Simple versioning system class diagram [9]***

Figure 34 presents the class diagram for implementing the security concern. All security functionalities are implemented in the `Security` class as operations namely `authenticate` and `authorize`. At the beginning of the working session, the user is authenticated by the operation `authenticate`. The successful result of this method is a `Security` object associated with the corresponding `User` object. These objects are maintained by the application for other authorization operations during the working session. A special interface namely `SecuredObject` is also needed to be implemented by classes whose objects are protected by authorization mechanism.

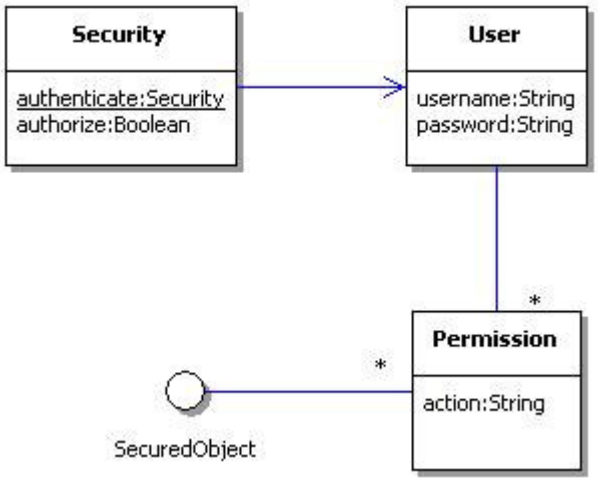



***Figure 34: Security class diagram***

Figure 35 shows that objects of `Product` and `Branches` classes are protected by the authorization mechanism. Each instance of these classes has permissions to be assigned explicitly to individual users. These classes also need to implement the `SecuredObject` interface.

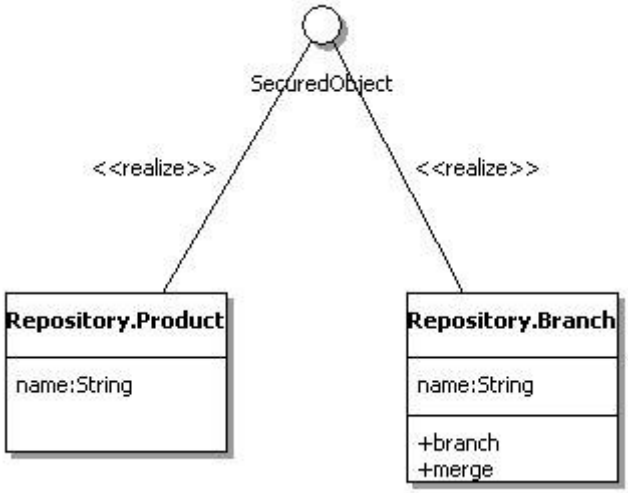



***Figure 35: Security realization***

The above class diagrams model the static structure of the system. The dynamic behavior of the system is modeled via diagrams like sequence diagrams, activity diagrams. Figure 36 is an example of the sequence diagram for the `branch` operation of the `Branch` class.

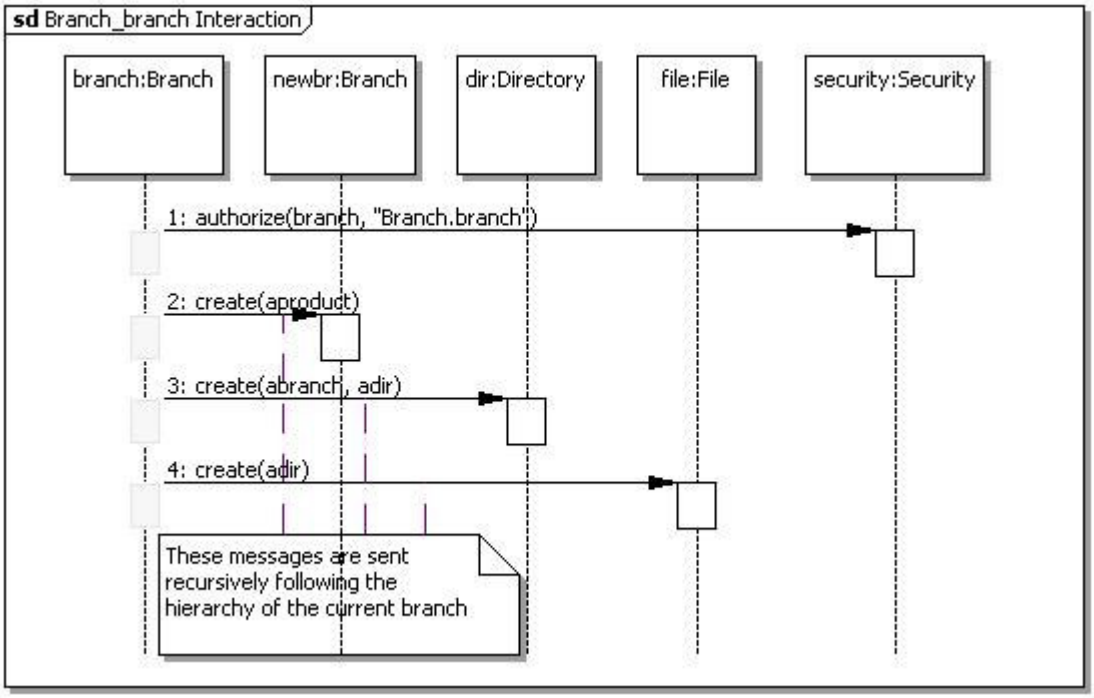



*Figure 36: Branching sequence diagram*

### 4.2.3 Relational PSM

Transformations from UML models to Relational models are typical examples that are usually introduced in QVT transformation languages. The specification of the QVT language in [23] also presents this example. Because of this, this subsection only describes the necessary steps to transform the PIM model of the CFVS to the corresponding PSM Relational model. A modification of the transformation definition in [23] then is provided.

Following is the necessary steps for the transformation; each step may be attached with a corresponding transformation rule given in the appendix:

1. For each persistent class in the UML model: (**`ClassToTable`**)
   a. Create a table with the same name in the Relational model.
   b. Create an ID column in the table.
   c. Create the primary key of the table using the ID column.
   d. For each primitive attribute of the class (**`PrimitiveAttributeToColumn`**):
      i. Create a field in the table with the same name. The type of the column corresponds to the primitive type of the attribute.
   e. For each complex attribute of the class, do the same steps 1d, 1e, 1f with the corresponding class type of the complex attribute (ComplexAttributeToColumn).
   f. Do the same steps 1d, 1e, 1f with the super class of the current class (SuperAttributeToColumn).
2. For each association between two persistent classes (source and destination) in the UML model (**`AssocToFKey`**):
   a. Identify the corresponding source and destination tables in the Relational model.
   b. Create a column in the source table for referring to the ID column of the destination table.
   c. Create a foreign key of the source table. The columns of this foreign key include only the column created in step 2b. The foreign key refers to the primary key of the destination table.

The transformation definition uses simplified metamodels of UML and Relational presented in Figure 28, Figure 29 and Figure 30. Table 6 below shows some of the transformation rules of this transformation definition, while Table 14 in the appendix presents the complete definition.

```
transformation umlRdbms(uml uses SimpleUML, rdbms uses SimpleRDBMS) {
key Table (name);
key Column (name, owner);
key Key (name, owner);
top relation ClassToTable {
    cn, prefix: String;
    checkonly domain uml c:Class {
        kind='persistent', name=cn
    };
    enforce domain rdbms t:Table {
        name = cn,
        column = cl:Column {name = cn + '_tid', type = 'NUMBER'} ,
        key = k:Key {name = cn + '_pk', column = cl}
    };
    where {
        prefix = ''; AttributeToColumn(c, t, prefix);
    }
}
relation AttributeToColumn {
    checkonly domain uml c:Class {};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
        PrimitiveAttributeToColumn(c, t, prefix);
        ComplexAttributeToColumn(c, t, prefix);
        SuperAttributeToColumn(c, t, prefix);
    }
}
...
top relation AssocToFKey {
    checkonly domain uml as:Association {
        name=asn,
        source=sc:Class {kind='persistent', name=scn},
        destination=dc:Class {kind='persistent', name=dcn}
    };
    enforce domain rdbms fk:ForeignKey {
        name=fkn,
        owner=srcTbl,
        column=fc:Column {name=fcn, type='NUMBER', owner=srcTbl},
        refersTo=k:Key {name= dcn + '_pk', owner = dstTbl}
    };
    when {
        ClassToTable(sc,srcTbl);
        ClassToTable(dc,dstTbl);
    }
    where {
        fkn = scn + '_' + an + '_' + dcn; fcn = fkn + '_tid';
    }
}
function PrimitiveTypeToSqlType(primitiveType:String):String {
    if (primitiveType = 'INTEGER')
    then 'NUMBER'
    else if (primitiveType = 'BOOLEAN')
        then 'BOOLEAN'
        else 'VARCHAR'
        endif;
    endif;
}
}
```

***Table 6: Some transformation rules from UML Model to Relational Schema***

As shown in Table 6, a transformation definition is declared by the keyword **`transformation`**, followed by a name and a list of model parameters with their metamodels.

A transformation definition consists of a set of transformation rules, each one is declared by a **`relation`** clause. However, not all relations will be executed by the engine when a transformation instantiated from this transformation definition, but only those which are declared with the keyword **`top`**. Non-top level relations will be invoked by top level relations and other relations. There are two top level relations in this example, namely **`ClassToTable`** and **`AssocToFKey`**.

The purpose of a relation is to keep the relationship between elements of the models participating in the relation hold. The participating models are declared by the **`domain`** clauses. For example, the relation **`ClassToTable`** is to keep the relationship between a **`Class`** in the UML model and a **`Table`** in the Relational Schema. Which domains are the sources and which one is the target will be specified during execution of the transformation. A domain may be specified as either **`checkonly`** or **`enforce`**. A relation execution with a checkonly domain as the target will only check the relationship and return an error if the relation does not hold. On the other hand, if an enforce domain is the target, the target domain may be modified in order to keep the relation hold; i.e. elements are created, deleted, or have their attributes be updated.

In order to search for instances of a participating domain in a model, keys may be specified using the **`key`** clauses. For example, instances of type **`Table`** will be identified uniquely by its attribute **`name`**, while instances of type **`Column`** need a combination of two of its attributes as the key, namely **`name`** and **`owner`**; i.e. the owning table.

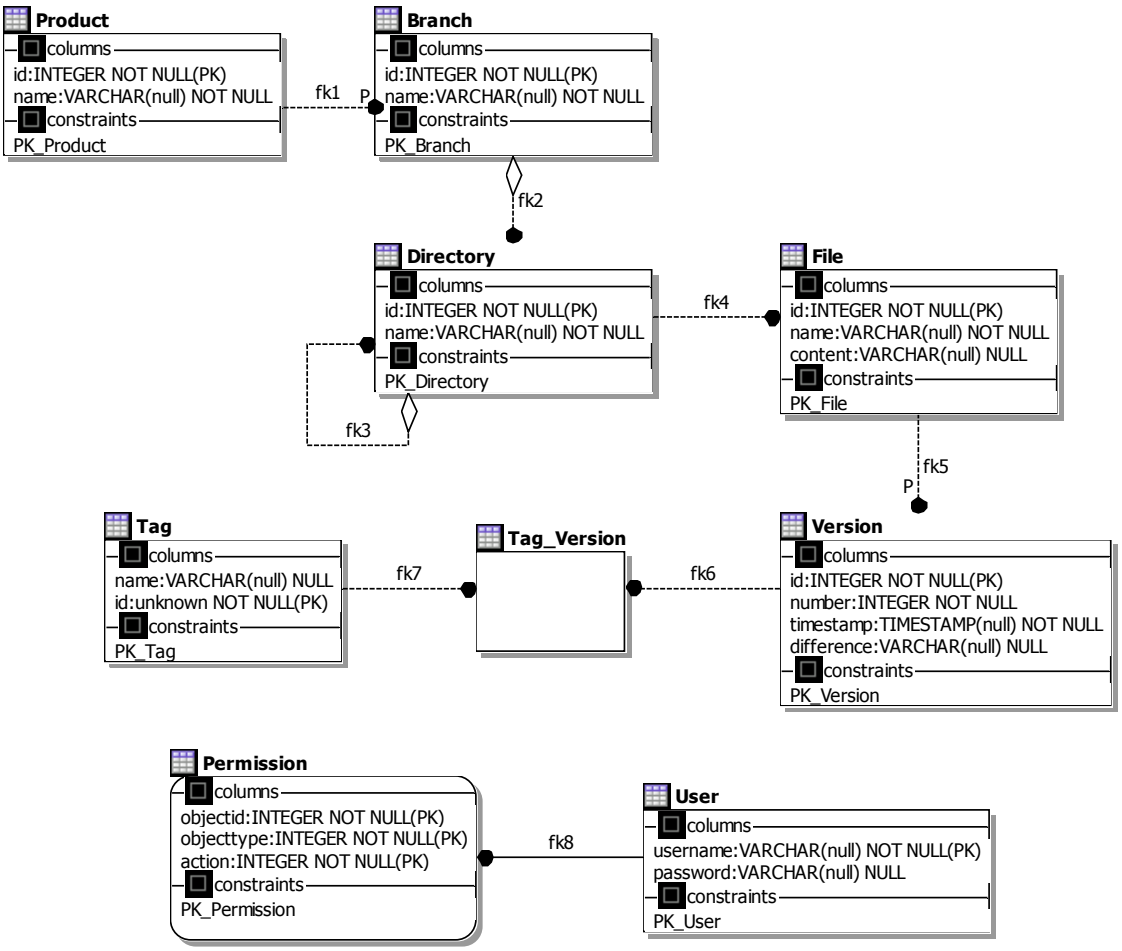



***Figure 37: CFVS Relational PSM Model***

A relation may be attached with a **when** and a **where** clause. The relation is only executed if its **when** clause holds and it is evaluated to true if the relationship between participating elements is kept and the **where** clause is true. For example, the relation **AssocToFKey** is only executed if the two classes in the association have been participated in a relation ClassToTable; i.e. the **when** clause, and the relation is true if both the main relationship and the **where** clause hold.

Besides keys and relations, a transformation definition may contain functions declared by **function** clauses. Functions are invoked inside the relations in order to evaluate a value from the input values. This example has a function **PrimitiveTypeToSqlType** which returns a String value representing the SQL type corresponding to the input UML type.

Figure 37 shows the entity relationship diagram of the Relational PSM model generated from the transformation discussed above.

### 4.2.4 Java PSM

Following are the steps to transform the UML model to the Java model:

1. For each package in the UML model, create the corresponding Java package in the Java model. The name of the Java package is defined by a tag value named JavaPackage of the UML package element (**UPackageToJPackage**).
2. For each class in the UML model (**UClassToJClass**):
   a. Create a Java class with the same name in the Java model in the corresponding Java package.
   b. For each attribute in the UML class (**AttributeToField**):
      i. Create a private field in the Java class. The field name has the form `m_<AttribName>`
      ii. Create the corresponding set/get methods.

a. Create methods in the Java class corresponding to the operations in the UML class (**OperationToMethod**).

3. Create the extends relation between Java classes in the Java model for each generalization relation in the UML model (**GeneralizationToExtends**).
4. Create corresponding fields and methods for Java classes in the Java model for each association relation in the UML model (**AssociationToField1**, **AssociationToField2**).
5. For each message in the sequence diagram from an instance of an UML class to another instance of another UML class, create an imports relation (if not existed) between two corresponding Java classes in the Java model (**MessageToImports**).

In order to implement this model transformation, metamodels of the source and target models are needed. A simple UML metamodel is presented in Figure 28 and Figure 29. Figure 31 shows the class contents of the UML Profile for Java, while Figure 32 shows the polymorphism and import relationship between Java classes [20].

The full transformation definition to transform UML models to Java models is provided in Table 15 in the Appendix B. The result of the transformation is partly shown in Figure 38.

## *4.3 Tools Support*

The case study in this thesis makes intensive use of the Borland Together Architect 2006 for Eclipse for modeling its models [6]. The tool helps to model the requirements model, the PIM model in the UML modeling language, and two PSM models for Relational and Java implementation.

The definition and execution of transformations are the major work in this case study and should be supported by a tool as well. However, this QVT language is so new that no tool has been ready to support it completely. Thus this work was done manually in this research.

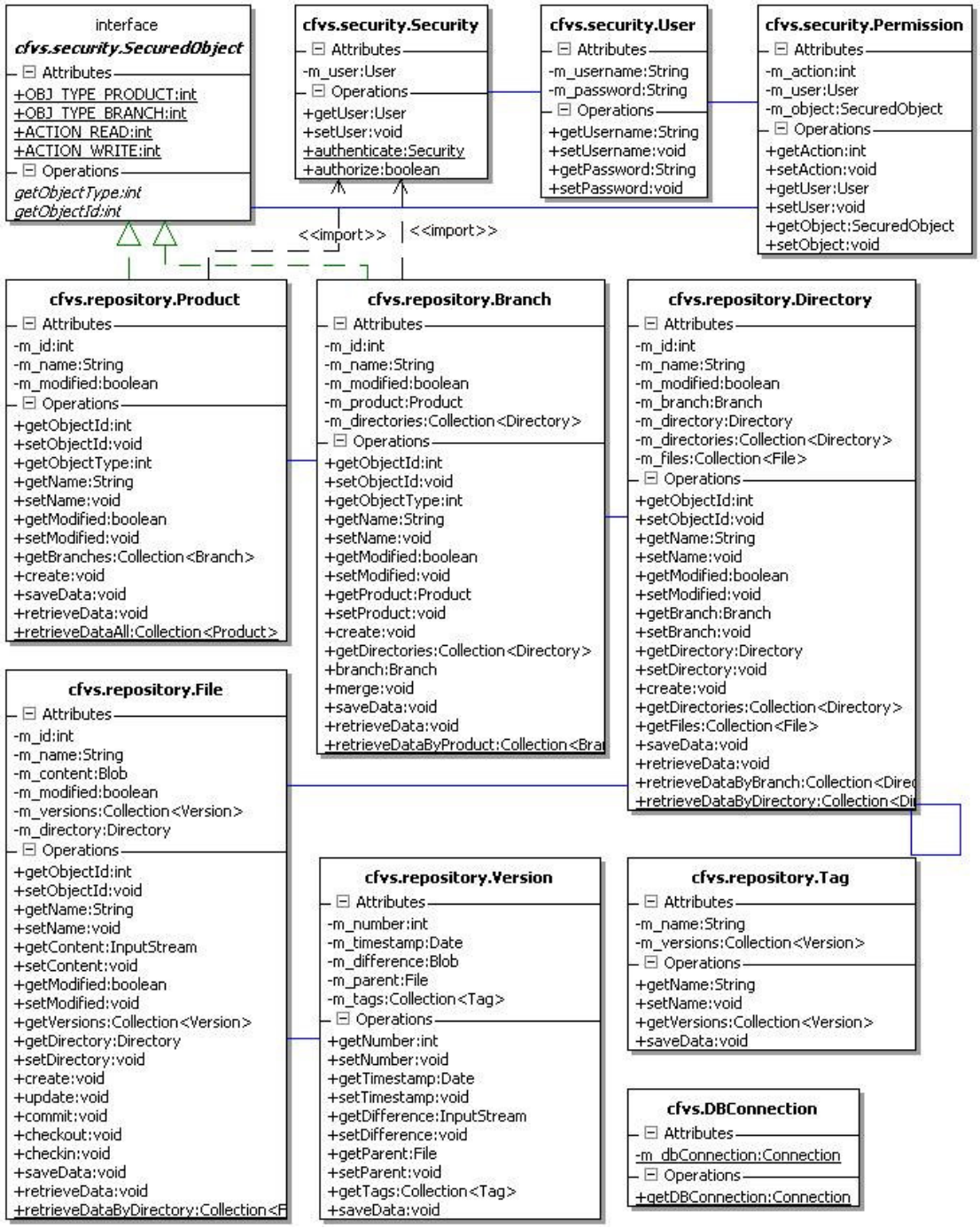



***Figure 38: Java classes***

The Borland Together tool allows writing transformation definitions in the imperative Operational Mappings language. We used this tool to write several transformation rules. However, due to some limitations, this work is merely for illustrative purpose. Firstly, the imperative Operational Mapping language is only an extension to the declarative languages to support imperative programming, thus it does not illustrate well the concepts of the QVT specification. Furthermore, the tracing model is incompletely implemented by the tool. Tracing classes and their instances are automatically deduced for transformation rules and transformation execution. However, only two fields or columns are allowed for each tracing class. This is unusable even for the simplest transformation. Finally, the thesis uses simplified versions of UML and RDBMS metamodels for analysis, namely SimpleUML and SimpleRDBMS, respectively. However, all models were developed in full versions of UML and RDBMS in the Borland Together. As such, the transformation definitions become very

complex due to the complexity of these metamodels, and thus we cannot define complete transformations of the case study with a limited time and effort.

There is another tool named ModelMorf developed by Tata Consultancy Services (TCS) [27]. The tool was claimed to fully support the QVT language. Unfortunately, we were unable to download the evaluation version of the tool, and thus we cannot use it for our case study.

The implementation of a transformation engine could be achieved by using graph transformations. Metamodels and models could be translated to type graphs and graphs which can be understood by the GROOVE tool [29]. Then transformation rules could be defined in the GROOVE tool to transform graphs to graphs. This kind of work is also available in the Master thesis of Nederpel [14]. However, this approach is not applied in this thesis project.

It is expected that several tools are ready in very near future to support the QVT language, and this research could be continued with respect to fully automatic transformations between models of the case study.

## 4.4 Summary

This chapter presented the case study developed in this research. The case study is a simplified Concurrent File Versioning System with basic functionalities such as checking in, checking out, tagging, branching and merging. Security and persistence requirements are also included. These are considered crosscutting concerns and will be used for analysis in the next chapter.

A process to implement the case study is extended from the MDA transformation pattern. It includes three main steps: building the PIM model, defining transformations to Relational and Java PSM models, and finally executing these transformations to derive these PSM models.

A discussion of the used tools was also provided. All models could be created with the Borland Together tool. However, it is unsatisfactory that the transformation work could not be done by the tool and has to be implemented manually. It is expected that this work will be completed in a further study with available tools in the near future.

The next chapter will use the developed transformation definitions to analyze the relation between crosscutting concerns in the case study, i.e. security and persistence, and the model transformations

# 5 Crosscutting in Model Transformations

This chapter analyzes the problem of crosscutting concerns in model transformations based on the case study introduced in the previous chapter. The goals of the analysis is to understand the problems that crosscutting concerns cause to model transformations and the ability of model transformations to help identify crosscutting concerns in a model.

Section 5.1 provides a decomposition analysis with respect to identifying concerns of the case study and types of elements in each phase of the development lifecycle. Section 5.2 uses the transformation definitions and the tracing information to derive dependency graphs at metamodel and model levels. These dependency graphs are helpful to identify several fundamental properties of the transformation with respect to crosscutting. Next, Section 5.3 discusses the problems that the crosscutting concerns in the case study cause to the model transformation. Finally, direct and indirect mappings of the derived dependency graphs are defined in Section 5.4, and summary is given in Section 5.5.

## *5.1 Decomposition Analysis*

A software product is developed through multiple phases, each phase has a different level of abstraction and an earlier phase involves models at a more abstract level. At each level, there are usually many possible ways of decomposition which may have a different affect on the quality of the final product, such as usability, performance, reliability, and on the problems of software development process itself, such as the problem of crosscutting that is being analyzed in this thesis.

In order to have a base point for analyzing the problem of crosscutting in the development of the case study, this section provides a preliminary analysis of the elements at different levels, namely requirements, use cases, design and implementation (both Java and Relational database). One of the possible ways of decomposition is selected for each level of abstraction to show which elements exist in the corresponding models and how they are related to corresponding concerns.

### 5.1.1 Concerns

The general concern of this case study is to develop a Concurrent File Versioning System. This concern is decomposed into a basic concern namely Repository and other extended concerns like Security, Persistence, and Transaction and so on. In the scope of this simple case study, following concerns are involved:

- *Repository*: this concern involves maintaining the repository for software artifacts during development. It is a basic concern representing the functionality of the CFVS product
- *Security*: this concern deals with management of users using this CFVS system and the rights each user may have to access different repository objects in the system. It is an extended concern and is considered to be a quality concern of the CFVS product.
- *Persistence*: this concern is about how the repository objects are stored in and retrieved from a storage media.

At the concern level, these concerns exist independently and do not depend on each other. In other words, these concerns are almost separated from each other.

### 5.1.2 Requirements and use cases

In this case study, requirements and use cases are considered to be at the same level of abstraction. Requirements are more text descriptive, while use cases are the corresponding elements specified in the UML modeling language.

Most of the requirements and use cases belonging to the Repository concern are described in sections 4.1.1 and 4.1.2, and modeled as use case diagrams shown in Figure 25 and Figure 26. Section 4.1.3 describes two of the most important requirements for the Security concern, namely authentication and authorization. Authentication is the requirement that a user must be

checked by a username and password pair before using the system, while authorization is the checking of whether a user is allowed to do an action on a particular object in the repository, for example, checking out a file.

| Requirements | Concerns |
|---|---|
| *Req1*: The system shall be able to maintain (i.e. add and remove) objects of different types of the CFVS application, such as products, branches, directories, files. | Repository |
| *Req2*: The system shall be able to check out and check in a file | Repository |
| *Req3*: The system shall be able to update and commit a file | Repository |
| *Req4*: The system shall be able to show the difference between two versions of a file | Repository |
| *Req5*: The system shall be able to create a new branch from an existing branch | Repository |
| *Req6*: The system shall be able to merge two branches | Repository |
| *Req7*: The system shall be able to authenticate a user before allowing her to use the application | Security |
| *Req8*: The system shall be able to authorize a user for appropriate permissions to manipulate particular objects in the application | Security |
| *Req9*: The system shall be able to store data of the CFVS application persistently in a storage media | Persistence |

***Table 7: Mapping requirements to concerns***

As identified, there are three concerns: Repository, Security and Persistence. The analysis of individual requirements shows that each requirement involves in only one of these concerns, as shown in Table 7. Thus, concerns at this level of abstraction are almost independent, or they are separated from each other.

### 5.1.3 Design (the PIM model)

As described in Section 4.2.2 and shown in the class diagrams in Figure 33, Figure 34 and Figure 35, elements of the design model of the case study include classes, their attributes and associations between them. These classes are grouped into two packages corresponding to the two concerns Repository and Security. Besides, the model uses tag elements which are considered as custom properties of other elements.

The model also starts modeling the behavior of the system by creating sequence diagrams for each significant method; usually the ones that realize directly the requirements, such as the methods `branch`, `merge` of the class `Branch`, or the methods `checkin`, `checkout` of the class `File`. There are several types of UML elements which are used in these sequence diagrams, such as Message, Lifeline. These are also considered to be elements of interest in this phase.

The `Repository` package includes classes `Product`, `Branch`, `Directory`, `File`, `Version`, and `Tag` which are kinds of objects in the CFVS system. These classes are related with each other mostly in an aggregation relation, as shown in Figure 33. Each class has its attributes and operations to implement various requirements of the Repository concern. For example, the `Branch` class has operations `branch` and `merge` for branching and merging requirements, while the `File` class has operations for other requirements like checking in, checking out and so on.

The most important class of the `Security` package is `Security` whose operations implement the requirements of the Security concern: `authenticate` and `authorize`. Other classes include `User` and `Permission` which represent user information and their rights to access repository objects. Finally, an interface `SecuredObject` is needed for any repository classes which need to be secured by the Security concern.

The elements of these two packages become strongly interdependent with each other. This is reflected in implementation relations between repository classes and the

`SecuredObject` interface as shown in the class diagram in Figure 35. The interaction between these two packages is also shown in the sequence diagrams of individual operations of repository classes in which messages are sent to a Security object; Figure 36 is an example of such a sequence diagram.

| UML elements | Concerns |
|---|---|
| Repository package<br>Repository classes: Product, Branch, Directory, File, Version, Tag<br>Attributes and operations of Repository classes<br>Associations between Repository classes | Repository |
| Security package<br>Security classes: SecuredObject, Security, User, Permission<br>Attributes and operations of Security classes<br>Associations between Security classes | Security |
| Implementation relations from Repository classes to Security classes<br>Call messages from Repository classes to Security classes | Repository, Security |
| Persistent Tags of Repository classes | Repository, Persistence |
| Persistent Tags of Security classes | Security, Persistence |

***Table 8: Mapping of the design elements (PIM) to concerns***

The third concern copes with the issue of storing and retrieving data for instances of different classes in a storage media – the Persistence concern. However, there are no separate classes for this concern. Instead, the concern is scattered in classes of other concerns by creating a special tag to mark persistent classes: all classes which have the custom property `kind` with value `persistent` will be persistent in the final CFVS system. In other words, elements of this concern are custom properties which are scattered in classes from other concerns. Table 8 shows different types of elements in the design model and how they are related to the three concerns.

### 5.1.4 Relational PSM model

As described, all classes marked as persistent in the PIM model are transformed to tables in the Relational PSM model. The tables in the Relational PSM model are persistent representatives of the classes in the PIM model. They are not divided into separate schemas for Repository and Security, but created in a single schema. The tables belonging to the Repository concern also include kinds of repository objects of the CFVS system, whereas the Security tables include only User and Permission. The table Permission represents the many-to-many relation between User and repository tables. Figure 37 in Section 4.2.3 shows this relation.

| Relational elements | Concern |
|---|---|
| Tables: Product, Branch, Directory, File, Version, Tag | Repository, Persistence |
| Tables: User, Permission | Security, Persistence |

***Table 9: Mapping Relational PSM elements to concerns***

### 5.1.5 Java PSM model

The Java PSM model is similar to the PIM model in that classes are divided into separate packages corresponding to the Repository and Security concerns. The Implementation relations in the PIM model become `implements` property, while the Dependency associations become `imports` property in the corresponding Java classes of the Java PSM model. In other words, the dependency of the Repository classes on the Security classes is embedded in the Repository classes as their properties, but not as separate elements. Figure 38 in Section 4.2.4 shows this relation.

| Java elements | Concerns |
|---|---|
| Repository package<br>Repository classes: Product, Branch, Directory, File, Version, Tag<br>Fields and methods of Repository classes | Repository |
| Security package<br>Security classes: SecurecObject, Security, User, Permission<br>Fields and methods of Security classes | Security |
| DBConnection class and its fields and methods | Persistence |
| `retrieveBy...` and `save` methods of Security classes | Security,<br>Persistence |
| `retrieveBy...` and `save` methods of Repository classes | Repository,<br>Persistence |

***Table 10: Mapping of the Java PSM elements to concerns***

The behavior of the Persistence concern becomes apparent in the Java PSM model by a separate class named `DBConnection` for handling database connection and by several operations in persistent classes namely `retrieveBy...` for retrieving data from database and `save` for storing modified data to database. However, the implementation of the Persistent concern is still embedded in classes of the other concerns. Table 10 shows elements of the Java PSM model and the mapping to corresponding concerns.

## 5.2 Dependency analysis

A transformation definition usually consists of multiple transformation rules with the issues of rule interaction and rule execution order. Rule interaction occurs when one rule needs to use the effects of another rule, such as creation, update or deletion of model elements; the most common case is to access a model element created by another rule. Rule ordering is the identification of the execution order among the rules in a transformation. These issues form some of the fundamental requirements for a transformation language [12].

Besides, there is a dependency between elements of source and target models with respect to the transformation between them. When there are changes to source elements, the changes are propagated to target elements by re-executing the transformation definition. It is required that the change propagation is performed incrementally, meaning that only transformation rules which are involved with the changed source elements need to be re-executed. This mechanism is implemented by maintaining the tracing information of the transformation execution [12][22][23].

This section proposes a method for deriving the dependency graphs at both metamodel and model levels based on the transformation definition and the tracing information of the transformation execution. Two transformation rules are selected. The definition of these rules written in the QVT language is presented and analyzed to see how elements in the source and target models are related to each other. Based on that information, individual dependency graphs are created and then combined. These dependency graphs will be used to draw conclusions with respect to rule interaction and rule execution order.

### 5.2.1 Transformation rules

**Transformation rule 1: `UClassToJClass`**

The goal of this transformation rule is to keep the relation between each class in the UML model with a unique Java class in the Java model. The tracing model for this transformation definition includes a QVT class for this rule which has two members of the types Class (imported from the UML metamodel) and JavaClass (imported from the Java metamodel). The definitions of the transformation rule and the tracing class are both provided in Table 11.

```
top relation UClassToJClass {
    ucn: String;
    checkonly domain uml uc:Class {name=ucn};
    enforce domain java jc:JavaClass {name=ucn};
    where {
        AttributeToField (uc, jc);
        OperationToMethod (uc, jc);
    }
}

class TUClassToJClass {
    uc: Class;
    jc: JavaClass;
}
```

***Table 11: UClassToJClass transformation rule and its QVT tracing class***

When executed with the Java model as the target model, the rule locates all classes in the UML model and, for each class, determines whether there exists a Java class with the same name in the Java model; if none exists, then the Java class is created. An instance of the tracing class is also created in the transformation engine for this pair of UML and Java classes.

**Transformation rule 2: MessageToImports**

```
top Relation MessageToImports {
    checkonly domain uml msg:Message {
        sendEvent = sendMsgEnd:MessageEnd {
            covered = sendLife:Lifeline {
                type = sendClass:Class {}
            }
        },
        receiveEvent = recvMsgEnd:MessageEnd {
            covered = recvLife:Lifeline {
                type = recvClass:Class {}
            }
        }
    };
    enforce domain java importingJClass:JavaClass {
        importsClass = importedJClass:JavaClass {}
    };
    when {
        UmlClassToJavaClass(sendClass, importingJClass);
        UmlClassToJavaClass(recvClass, importedJClass);
    }
}

class TMessageToImports {
    msg: Message;
    sendMsgEnd: MessageEnd;
    recvMsgEnd: MessageEnd;
    sendLife: Lifeline;
    recvLife: Lifeline;
    sendClass: Class;
    recvClass:Class;
    importingJClass: JavaClass;
    importedJClass: JavaClass;
}
```

***Table 12: MessageToImports transformation rule and its QVT tracing class***

The purpose of the transformation rule `MessageToImports` is to transform call messages modeled in interaction diagrams, such as the sequence diagram shown in Figure 36, of the UML model to corresponding elements in the Java model. The ideal transformation is to create appropriate statements in the Java method corresponding to the UML operation that is described by the interaction diagram. However, due to the limited modeling capability of the Java metamodel used in this case study, the transformation rule only creates the `imports` relation between two Java classes corresponding to the UML classes participating in the call message.

The tracing class for this transformation rule is shown in Table 12. It includes members for the Message element, the UML Class elements participating in the Message element, and the corresponding JavaClass elements.

### 5.2.2 Dependency graphs at metamodel level

It is straightforward to derive the dependency graph between source and target elements at the metamodel level for the first rule as shown in Figure 39. In this figure, the upper eclipse represents the source metamodel, while the lower one represents the target metamodel. Each box in these eclipses represents an element of a specific type as defined in the metamodel that the rule refers to. A line between a source element and a target element represents the relation between them as derived from the definition of the rule.

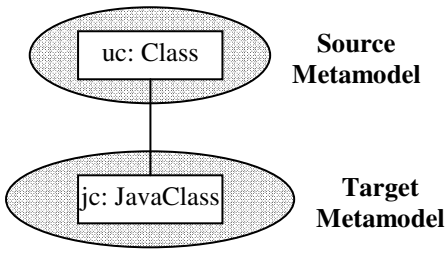



***Figure 39: Dependency graph of UClassToJClass at metamodel level***

There are several options for deriving the dependency graph for the second rule. One possible option is to let every source elements be mapped to every target elements involved in the rule. The result may be described by Figure 40 below.

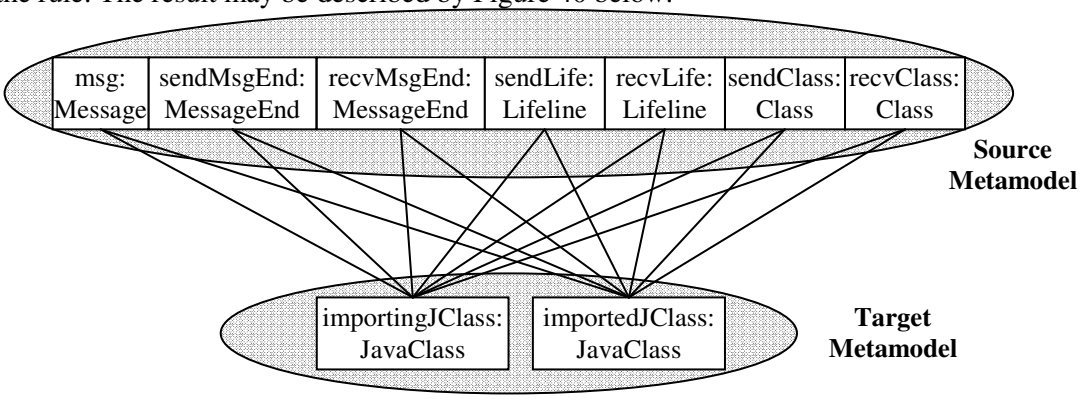



***Figure 40: Dependency graph of MessageToImports at metamodel level (initial derivation)***

However, this mapping derivation does not reflect the fact that when there are changes to the source elements, only the `importingJClass`, but not the `importedJClass`, needs to be changed with respect to this transformation rule. Thus the mappings from the source elements to the `importedJClass` in Figure 40 are not appropriate and should be ignored.

In order to solve this issue, the allocation of variables of the rule written in the Core language could be used. As discussed in Section 2.4.3, the QVT language specification provides a set of rules to translate automatically a transformation rule written in the Relations language to another one written in the Core language which groups variables of the rule into different areas and patterns. Figure 41 shows the allocation of variables for the second rule. It is observed from this allocation that only the target element `importingClass` depends on

the source elements `msg`, `sendMsgEnd`, `recvMsgEnd`, `sendLife`, `recvLife`, `sendClass` and `recvClass`.

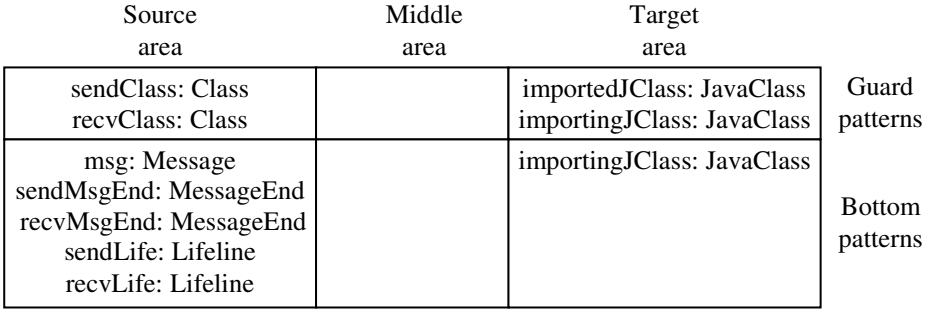



***Figure 41: Allocation of variables of the rule MessageToImports***

In addition, there is a difference among the mapping relations from source elements to target elements in terms of change effect. With respect to this rule, any change to `msg`, `sendMsgEnd`, `recvMsgEnd`, `sendLife or recvLife` will cause immediate changes to `importingJClass`, or the change is direct. On the other hand, when there is a change with either `sendClass` or `recvClass`, this change is possibly propagated to `sendLife`, `recvLife`, `sendMsgEnd`, `recvMsgEnd and msg`, and consequently to the target element `importingClass`; this kind of change is considered indirect. Because of this fact, mapping relations between source and target elements are classified into two types as described below, and their formal definitions are presented in Section 5.4.

- *Direct mapping*: a direct mapping of a transformation rule is a mapping between a source element and a target element in which any change to the source element would make direct changes to the target element with respect to this rule.
- *Indirect mapping*: an indirect mapping is a mapping between a source element and a target element in which a change to the source element would make changes to other source elements of the same rule and subsequently make changes to the target element.

Note that this classification into direct and indirect mappings is applied to mapping relations at both metamodel and model levels. At metamodel level, the mappings are between elements of the source and target metamodels in the definition of the transformation rule. At model level, the mappings are between source and target elements which participate in an application of that transformation rule; i.e. they appear in an instance of the tracing class corresponding to that transformation rule.

At metamodel level, the determination of whether a mapping is direct or indirect is also based on the Core pattern box as shown in Figure 41. The mappings from the source elements in the bottom pattern of the source area to the target elements in the bottom pattern of the target area are direct, while the mappings from the source elements in the guard pattern of the source area are indirect.

Based on the definition of mapping types and the method to determine the type of a mapping, dependency graph at the metamodel level of the rule MessageToImports could be derived as shown in Figure 42.

Figure 39 and Figure 42 show the dependency graphs at the metamodel level for the two transformation rules introduced in the previous section. Each dependency graph should contain direct mapping relations which represent the effects of the rule, whereas it may contain zero or more indirect relations. Each indirect relation between source and target metamodel elements means that the rule interacts with another rule which contains the same type of source element. For example, the graph dependency for the rule MessageToImports contains several direct mappings between a set of UML elements and a Java JavaClass element; these direct relations mean that the existence of a set of {Message, MessageEnd, MessageEnd, Lifeline, Lifeline} elements in the source model will have an immediate effect on a JavaClass element in the target model. The graph also contains indirect mappings between UML Class elements and a Java JavaClass element, thus the execution of the rule

would access some elements created by the execution of another rule (UClassToJClass in this case) on the same Class elements.

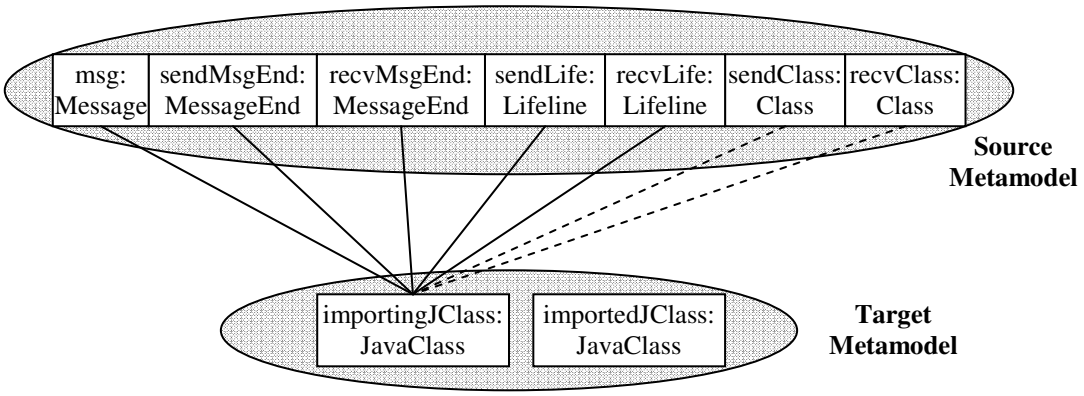



***Figure 42: Dependency graph of MessageToImports at metamodel level (accepted derivation)***

These dependency graphs are also useful as a hint for identifying transformation rule conflicts. Transformation rules conflict with each other when two different source metamodel elements have a same type of effect on a single target metamodel element. In the above example, both UML metamodel elements Class and Message have a direct mapping with Java metamodel element JavaClass, but they create different types of effect - one creates the JavaClass element, while the other updates one of its attributes which is not involved in the first rule. However, if for example there is a transformation rule which transforms association elements in the UML model to `imports` relation in the Java model, this rule would conflict with the MessageToImports rule, as both rules update (i.e. setting or clearing values of) the `importsClass` attribute of JavaClass elements in the Java model. In this case, a desired effect of one rule may create undesired effect for the other rule for any order of execution, so the rules should be redesigned appropriately.

In summary, the dependency graph at the metamodel level is a good facility for identifying rule interaction and rule execution ordering. Based on these dependency graphs, the transformation rules in a transformation definition could be assessed conveniently and the rules could be redesigned appropriately.

### 5.2.3 Dependency graphs at model level

With the specification of the classes in the UML PIM model shown in Figure 33 and Figure 34, the dependency graph for the first rule could be generated as shown in Figure 43. One or more numbers are attached to each mapping relation to mark the rule number and the number of the tracing data record representing a specific application of this rule; this number is called the rule application number.

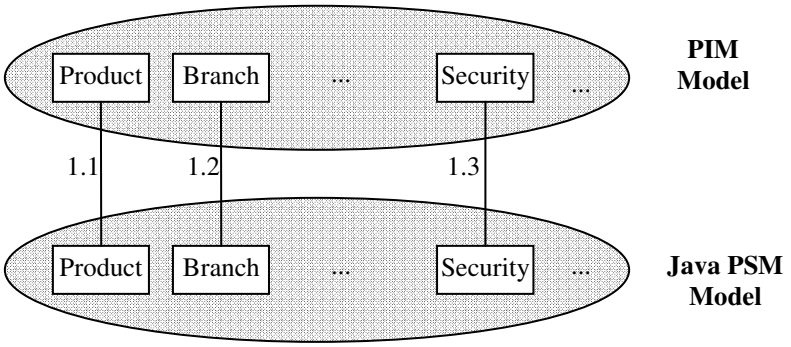
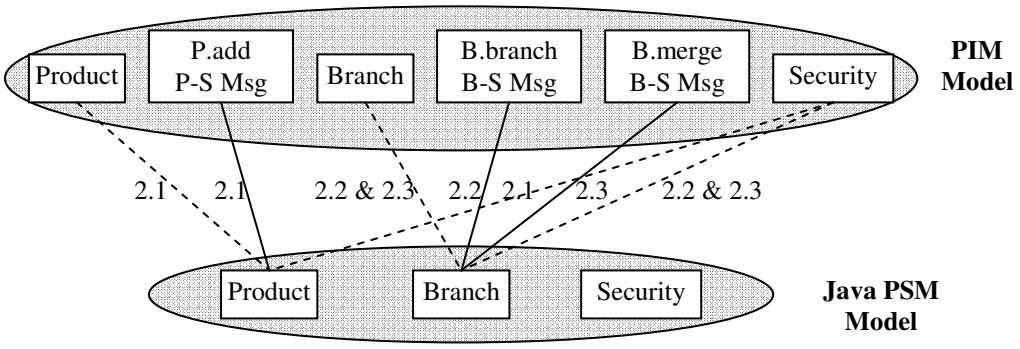



***Figure 43: Dependency graph of UClassToJClass at model level***

In the UML PIM model described in Section 4.2.2, several UML operations are modeled as sequence diagrams in which several Message elements are created to show the usage of the Security classes by the Repository classes. Figure 36 is such a sequence diagram. Based on the mapping derivation method above, the transformation execution of the rule MessageToImports generates the dependency graph as shown in Figure 44.

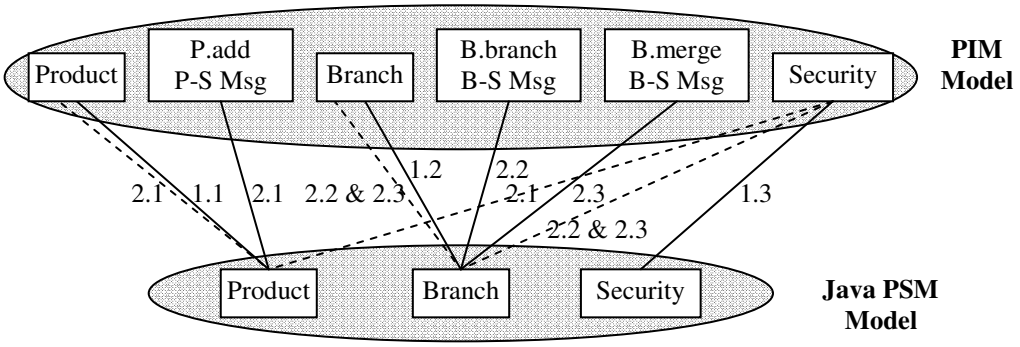



**Note:** Some elements are omitted for clearer view

***Figure 44: Dependency graph for the rule MessageToImports***

These individual dependency graphs help to identify when a specific application of a rule needs to be re-applied (or re-execution of the rule on a specific set of source and target elements). For example, assume that there is a single change to the source element "`P.add P-S Msg`", the rule application 2.1 needs to be re-executed. However, when the source element `Product` changes, besides the re-execution of the rule application 1.1, the rule application 2.1 is re-executed only when this change also makes changes to the source element "`P.add P-S Msg`".

When there are multiple changes to the source model, multiple rule applications need to be re-executed. In this case, the combined dependency graph should be used to identify the order of re-execution of these rule applications. Figure 45 is the combination of the two dependency graphs introduced above and Table 13 shows the corresponding dependency matrix for a clearer view of direct and indirect mappings between source and target elements.




--------- Indrect mapping relation

———— Direct mapping relation

1.*: Mapping relation from UClassToJClass

2.*: Mapping relation from MessageToImports

**Note**: Some elements and mappings are omitted for clearer view

***Figure 45: Combination of two dependency graphs***

When the source element "`P.add P-S Msg`" is modified, the rule application 2.1 needs to be re-executed. However, this rule application interacts with other rule applications because it contains indirect relations from source elements `Product` and `Security`. If the source element `Security` is changed as well, then the rule application 1.5 needs to be re-executed first. In general, it is possible to order the execution of rule applications based on this

combined dependency graph as follows: any of the rule applications in the group (1.1, 1.2, 1.3) should be executed before any of the rule applications in the group (2.1, 2.2, 2.3); rule applications in the same group can be executed in any order.

| **Target**<br>**Source** | Product | Branch | Security |
|---|---|---|---|
| Product | 1.1 direct<br>2.1 indirect | – | – |
| P.add P-S Msg | 2.1 direct | – | – |
| Branch | – | 1.2 direct<br>2.2 indirect<br>2.3 indirect | – |
| B.branch B-S Msg | – | 2.2 direct | – |
| B.merge B-S Msg | – | 2.3 direct | – |
| Security | 2.1 indirect | 2.2 indirect<br>2.3 indirect | 1.3 direct |

***Table 13: Dependency matrix for the combined dependency graph***

In conclusion, the specification of transformation rules and the tracing information of the execution of these rules could be used to generate the individual and combined dependency graphs. These dependency graphs are helpful in identifying which rule applications need to be re-executed in which order when there are changes to the source model; this is a key issue to implement the incremental model compilation of a transformation language.

## 5.3 Crosscutting Concerns Analysis

Crosscutting concerns are known to cause several problems to various artifacts of the software development. In this section, the transformation of the case study is analyzed to see how it is affected by the identified crosscutting concerns.

Crosscutting is not an inherit property of concerns, but it depends on the context of the software development, such as the environment in which these concerns are realized or the dominant decomposition of the problem. For example, the persistence concern would not be crosscutting in the EJB environment in which necessary services are provided to realize this concern transparently to other concerns, but it becomes crosscutting in a standalone Java application. In the context of this case study to develop a client-server Java application, Persistence and Security are the crosscutting concerns.

These concerns are involved in transformation rules. The previous section presents two transformation rules `UClassToJClass` and `MessageToImports`. The first rule transforms a UML class to Java class; the class may realize one of the identified concerns, as shown in the decomposition analysis of Section 5.1. The derived dependency graphs at both metamodel and model levels for this rule contain no indirect mappings. Thus the rule does not interact with any other rule, and the rule can be executed without consideration of the order.

The situation is more complex for the second rule. The rule transforms message calls in the UML model to corresponding `Imports` property of Java classes. As presented in Section 5.1, the UML design model contains Message elements at which a method belonging to the Repository concern calls the `authorization` method belonging to the Security concern. According to the definition of crosscutting in Section 3.3 and as presented with the combined dependency graph in Figure 45, it is considered that the Security class crosscuts these `Message` elements. As a consequence, the dependency graphs for the rule `MessageToImports` becomes more complex. It contains several indirect mappings with the meaning that the rule interacts with some other rule (in this case, the other rule is `UClassToJClass`). At the execution time, a specific application of the rule should be executed after the application of the rule `UClassToJClass` for the involved UML classes (either `sendClass` or `recvClass`, or both) if these classes are changed.

Besides the problems with these specific properties, crosscutting concerns also cause general problems, such as complexity and change impact, to model transformations. However, these concerns cause different degrees of severity in different ways.

In the design model, the Security concern is realized by a separate package and a set of interaction diagrams to show how the concern interacts with other concerns. Apparently, the model becomes much more complex, as these interactions are scattered to almost everywhere; each method of `Repository` classes sends a message to the `Security` class to check for execution permission. In this case, the Security concern causes high impact of change to the design model. Whenever the Security requirements change, besides modification to elements of the `Security` package, elements in the `Repository` package may need to be modified as well. For example, a new requirement for logging user activities would require changes to all sequence diagrams of methods of Repository classes where interaction with the `Security` class occurs.

However, the concern causes lower impact of change to the transformation rules. For example, the rule `MessageToImpacts` would not consider much with the new requirement of logging user activities; it is only needed to re-execute the rule to transform the newly added `Message` elements for these logging activities.

On the other hand, the Persistence concern does not cause much problem to the design model. Each class that is involved with persistence is marked by a custom property, thus model does not become complex because of this concern. Any change to the elements of the Persistence concern would not create any impact on the rest of the model.

However, the place where the Persistence concern causes more problems is the transformation definition. The case study contains several transformation rules in which all classes marked with persistent custom property are transformed to Java classes with necessary methods for retrieving data from and saving data to storage media. Apparently, these rules become much more complex as they use not only information from source and target metamodels, but also information on how to realize the Persistence concern. The situation is worse when this information is represented in a textual form but not modeled in a graphical model as in MDA development framework. This consequently leads to the difficulty in reusing the rules in other applications involved with the Persistence concern. The changes in the Persistence concern also cause these rules modified instead of only re-executing them, and have an impact on other dependent rules.

In summary, crosscutting concerns are the cause of several problems to QVT model transformations. Some are related to specific properties of model transformations, namely transformation rule interaction and execution order of transformation rules. Others are general problems, such as high complexity and change impact.

## 5.4 Definition of Direct and Indirect Mappings

This section provides the definitions of direct and indirect mappings of a dependency graph as introduced in Section 5.2 based on the definition of transformation rules written in the Core language of the QVT specification.

A transformation rule defines a relation that should be maintained between source and target models. It consists of a set of variables and constraints or derivations which are divided into areas and patterns as shown in Figure 46 below:

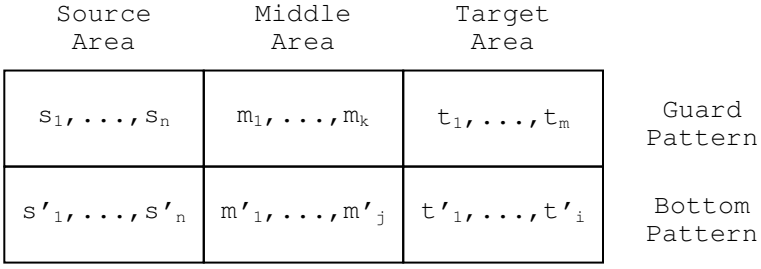



***Figure 46: Allocation of variables of a transformation rule to areas and patterns***

$s_1, \ldots, s_n$ are guard pattern variables to be matched to elements of the source model.

$t_1, \ldots, t_m$ are guard pattern variables to be matched to elements of the target model.

$m_1, \ldots, m_k$ are the guard pattern variables of the middle area of the transformation.

$s'_1, \ldots, s'_n$ are bottom pattern variables to be matched to elements of the source model.

$t'_1, \ldots, t'_i$ are the bottom pattern variables to be matched to elements of the target model.

$m'_1, \ldots, m'_j$ are the bottom pattern variables of the middle area of the transformation.

There are several notes in this figure:

- Each pattern, besides its variables, also contains constraints or derivations. Constraints are checked to ensure that a matching of the pattern's variables to the corresponding model elements is valid. Derivations are enforced; i.e. the pattern's variables are realized by creating and/or updating elements in the corresponding model to make the matching become valid.
- The binding of elements to guard pattern variables $s_1, \ldots, s_n$, $t_1, \ldots, t_m$ and $m_1, \ldots, m_k$ is not executed by the transformation engine, but is provided by the invoking transformation rule. This binding is provided as the context for the matching of elements to the bottom pattern variables; i.e. these variables are used in the constraints and derivations of the bottom patterns.
- The binding of elements to bottom pattern variables is executed by the transformation engine to ensure that there exists at least one matching to target bottom pattern variables $t'_1, \ldots, t'_i$ for each matching to source and middle bottom pattern variables $s'_1, \ldots, s'_n$ and $m'_1, \ldots, m'_j$.

See Figure 16 for a concrete example of the allocation of variables to areas and patterns.

The transformation rule is then defined by a procedure with the following signature:

Rule: $(s_1, \ldots, s_n, s'_1, \ldots, s'_n) \rightarrow (t'_1, \ldots, t'_i)$

This signature means that a tuple of elements $(t'_1, \ldots, t'_i)$ is derived in the target model from a tuple of elements $(s_1, \ldots, s_n, s'_1, \ldots, s'_n)$ in the source models. The variables $(t_1, \ldots, t_m)$ of the guard pattern of the target area are not in the signature because they are not derived with respect to this rule. They are actually derived by other transformation rules which this rule interacts with. The variables $(m_1, \ldots, m_k)$ and $(m'_1, \ldots, m'_j)$ in the middle area are also not in the signature because they do not belong to any model, but local to the transformation definition.

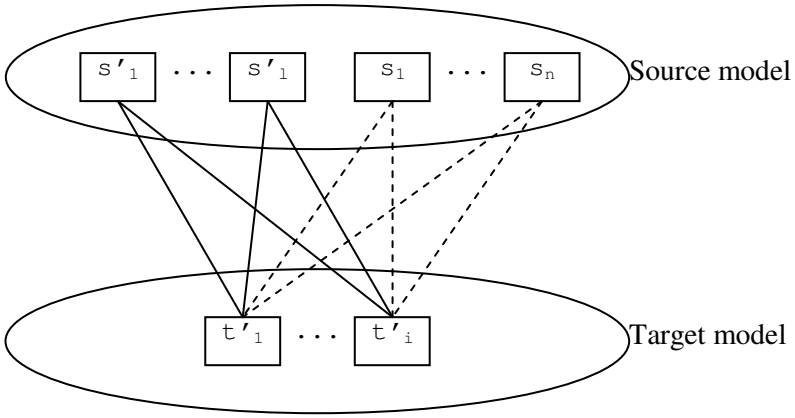



***Figure 47: The derived dependency graph of the transformation rule***

In addition, the variables of the guard pattern $(s_1, \ldots, s_n)$ and the variables $(s'_1, \ldots, s'_n)$ of the bottom pattern of the source area have different roles with respect to the change impact on the variables $(t'_1, \ldots, t'_i)$ of the bottom pattern of the target area. Any change to $(s'_1, \ldots, s'_n)$ would cause this rule to be re-executed immediately, while a change to $(s_1, \ldots, s_n)$ does not have a direct impact on $(t'_1, \ldots, t'_i)$, but that change is

possibly propagated to ($s'_1, \ldots, s'_n$) (because of coupling relations in the source models) and consequently to ($t'_1, \ldots, t'_i$) (because of this rule).

Based on this information, the dependency graph for the transformation rule could be derived as shown in Figure 47. The dependency graph includes direct (represented by solid lines) and indirect (represented by dashed lines) mappings between source and target elements. These direct and indirect mappings are defined as follows:

- A *direct mapping* is defined as the mapping from each variable of the bottom pattern of the source model ($s'_1, \ldots, s'_n$) to each variable of the bottom pattern of the target model ($t'_1, \ldots, t'_i$).
- An *indirect mapping* is defined as the mapping from each variable of the guard pattern of the source model ($s_1, \ldots, s_n$) to each variable of the bottom pattern of the target model ($t'_1, \ldots, t'_i$).

## 5.5 Summary

This chapter made a detailed analysis of the impact of crosscutting concerns on QVT model transformations based on the case study. The first part introduced concerns and types of elements existing in each phase of the development of the CFVS system. Two concerns which are considered as crosscutting are identified: security and persistence.

The second section chose two transformation rules from the UML to Java transformation definition. Dependency graphs for these transformation rules are derived at both metamodel and model levels. The derivation is based on the specification of the rule in the Core language (for metamodel level) and the tracing information (for model level). These dependency graphs are used to identify several properties of model transformations.

The next section discussed the problem that the crosscutting concerns security and persistence cause to the model transformation of the case study. Some problems are related to specific properties of the transformation, namely transformation rule interaction and execution order of transformation rules. Others are general problems, such as high complexity and change impact.

The final section provided a definition of direct and indirect mappings of the dependency graphs according to the derivation method proposed in this study. The definition is based on the allocation of variables to areas and patterns of transformation rules written in the Core language.

The major observations with this analysis are that crosscutting concerns cause as much problem to model transformations as to other artifacts of the software development. However, model transformations provide a method to identify crosscutting concerns by deriving dependency graphs based on traceability. Another observation is also drawn from the case study is that model transformations based on this QVT language could be well applicable to models in which a MOF-compliant AO modeling technique is applied. The final chapter will provide conclusions and discussions based on this analysis and these observations.

# 6 Conclusion

This chapter summarizes the main results of this thesis. Section 6.1 gives a summary of the content of the thesis before conclusions and discussions of the research are provided in Section 6.2. The final Section 6.3 provides several recommendations and directions for further research based on this study.

## 6.1 Summary

Chapter 2 provided a general introduction to the Model Driven Architecture framework and an explanation of the proposed QVT language for model transformations. The language includes several sub-languages. The sub-languages Relations and Core are two declarative languages with the same semantics at different levels of abstraction and the imperative language Operational Mappings provides a method of extension to these declarative languages with imperative operations.

Relations is a user-friendly metamodel and language which supports complex object pattern matching and object template creation. Traces between elements involved in a transformation are generated implicitly. The Core metamodel and language is defined using the minimal extensions to EMOF and OCL. Trace classes are defined explicitly as MOF models, and trace instances are created or deleted in the same way as creation and deletion of any other objects. Finally, the chapter discussed in detail the concepts used in the Relations and Core languages by concrete examples.

Chapter 3 provided an introduction to the Aspect Oriented Software Development methodology and several frameworks to define the concepts of scattering, tangling and crosscutting within AOSD. In a framework proposed by Masuhara and Kiczales, the core semantics of the aspect-oriented mechanism was captured by modeling the weaving process. The authors then defined the concept of crosscutting: for a pair of modules $m_A$ and $m_B$ (from $p_A$ and $p_B$), mA is said to crosscut mB with respect to X if and only if their projections onto X intersect, and neither of the projections is a subset of the other.

Berg and Conejero enhanced this framework by providing another conceptual framework in which the concepts of scattering, tangling and crosscutting are defined explicitly in terms of a source with respect to a target. The relationship between the source and the target was represented by a Crosscutting Pattern in which elements in the source were mapped to elements in the target. Based on this crosscutting pattern, concepts of scattering, tangling and crosscutting could be defined explicitly as different situations of the mapping relations. This framework was chosen to use in this study as (1) it provided explicit definitions for scattering, tangling and crosscutting, and (2) it used the traceability between a source and a target; this is a basic property of QVT-based model transformations.

Chapter 4 presented the case study: a Concurrent File Versioning System. A CFVS system allows the user to keep multiple versions of files ordered by a timestamp. The user may get an existing version of a file from the system, modify that file and save it back to the system as a new version without discarding the previous version of that file. Files are organized in a directory hierarchy. The chapter introduced requirements of the CFVS system. The functional requirements include basic functionalities such as checking in files, checking out files, committing a file or updating a file, and extended functionalities. Quality requirements include security like authentication and authorization, and persistence.

In this case study, we built the PIM model, or the design model, by the UML modeling language. Transformation rules written in the Relations language were then defined. The aim of this transformation definition is to transform the PIM model to automatically generate a Relational and a Java PSM model. However, due to the unavailability of such a transformation execution engine, the transformation execution could only be done manually. Despite of this limitation, the chapter still provided a complete QVT transformation which would be used as the basis for the analysis in the next chapter.

Chapter 5 analyzed different aspects of the transformation in the case study with respect to crosscutting concerns. The chapter included a decomposition analysis, a dependency analysis and a discussion of problems that crosscutting concerns cause to the case study.

In the decomposition analysis, we provided a preliminary analysis of the elements at different phases of the development of the case study, namely requirements, use cases, design and implementation. One of the possible ways of decomposition was selected for each level of abstraction to show which elements existed in the corresponding models and how they were related to corresponding concerns.

The dependency analysis presented an approach to derive the dependency graphs at the metamodel and model levels, and how these dependency graphs were used to identify various properties of the transformation. The dependency graph for each transformation rule at the metamodel level contains direct mappings which represent the effects of the rule, whereas it may contain zero or more indirect mappings. Each indirect mapping between source and target metamodel elements means that the rule interacts with another rule which contains the same type of source element.

The dependency graph at the model level is derived based on the tracing information of the execution of the transformation rules. These dependency graphs are helpful in identifying which rule applications need to be re-executed in which order when there are changes to the source model; this is a key issue to implement the incremental model compilation of a transformation language.

Finally, the chapter provided a discussion on the problems that crosscutting concerns caused to the different elements of the case study. The security concern caused more problems to the design model, while the persistence concern had a severer impact on the transformation definition to transform this design model to the implementation models.

## 6.2 Discussion

In this thesis, we aim at analyzing the impact of crosscutting concerns on model transformations based on the QVT transformation language. Following are our conclusions in this study:

***Traceability of QVT model transformations can be used to identify crosscutting concerns.***

Chapter 3 introduces the definition of concepts of scattering, tangling and crosscutting in terms of “one thing” with respect to “another thing”. This definition is represented by a mapping relationship, or a dependency graph, between elements of the source and the target models in a model transformation based on this QVT language.

A dependency graph for an individual transformation rule could be derived at the metamodel level based on the definition of the rule itself, while another dependency graph is derived at the model level based on the tracing information of the execution of that rule. Dependency graphs for the whole transformation are derived by combining these individual dependency graphs at both metamodel and model levels, respectively. Section 5.2 presents the method for this derivation.

Based on the decomposition analysis in Section 5.1, these dependency graphs are studied to identify the crosscutting behavior between source elements with respect to the target elements. However, there is a tendency that every element crosscuts other elements as the dependency graphs become very large and complex in a real-life transformation. Thus these dependency graphs should be pre-processed before they could be used for identifying crosscutting effectively. One possible method could be to distinguish between direct and indirect mappings between source and target elements and then to consider only the direct ones, as discussed in Section 5.2.

In conclusion, the transformation rules and corresponding tracing information can be used to derive dependency graphs in order to identify the crosscutting behavior; however, these dependency graphs should be further processed before they become really useable in a real-life application. Section 6.3 provides several recommendations for this processing.

***Crosscutting concerns causes several problems on QVT model transformations; these include:***

***(a) General problems, such as complexity and change impact***

Section 5.3 shows that crosscutting concerns cause problems of complexity and change impact to model transformations; i.e. transformation rules involved with a crosscutting concern is more complex than the others and when that crosscutting concern changes, they may be changed as well.

The severity of the problem depends on how the concern is realized. If the concern is realized explicitly at the source model level, then the transformation definition does not need to consider much about the concern; as a result, the complexity and change impact is low. However, some designers may decide to model the concern implicitly at the source model level by a little piece of information, such as the tag to mark persistent classes presented in the case study of chapter 4, and write a complex transformation definition to realize this concern, the change impact of this concern on the transformation becomes higher.

***(a) Specific problems to properties of model transformations, namely transformation rule interaction and execution order of transformation rule***

Section 5.2 and Section 5.3 also show that crosscutting concerns cause some problems on model transformations with respect to their properties: transformation rule interaction and execution order of transformation rules. Rule interaction occurs when one rule needs to use the effects of another rule, and rule ordering is the identification of the execution order among the rules in a transformation.

When a crosscutting concern is realized by a transformation rule, the dependency graph for the rule becomes more complex and involves more indirect mappings. The problem becomes much worse when individual dependency graphs are combined to create the dependency graph for the whole transformation. This means that the transformation rules of the transformation are more interdependent with each other.

When the transformation rules are highly interdependent, the determination of the execution order of transformation rules becomes more difficult and less efficient. This determination is necessary to have a high performance of transformation execution and it is a requirement for implementing incremental compilation of target models.

In conclusion, crosscutting concerns cause problems of change impact, higher interdependent transformation rules and more difficult to determine the execution order of transformation rules. These problems make model transformations more complex, less robust, less reusable and less adaptive.

## 6.3 Recommendations and Future Work

Following are our recommendations based on the result of this research:

**Traceability extension**

Traceability in QVT model transformations is used to implement incremental compilation of target models, as described in OMG's Request for Proposal. However, this property should be studied and extended to deal with the problem of crosscutting concerns. One possible extension is to follow the approach of the Core language in which variables of a transformation rule are allocated to guard and bottom patterns. By that approach, the role of each element involved in a transformation rule application can be identified immediately; i.e. whether a source element has a direct or indirect effect on target elements, or whether a target element is a derived one or an existing one which is derived by another rule application.

There are several benefits to this extension. Firstly, the rule interaction and rule execution order properties are reflected in the tracing information. Thus the individual and combined dependency graphs with the distinction of direct and indirect mappings at both metamodel and model levels can be derived immediately without considering the definition of transformation rules written in the Core language. This benefit leads to the second one that the derivation process of dependency graphs does not depend on the specific specification of the

Core language; this means that transformation rules can be written in any QVT languages with the requirement that the traceability of transformations is maintained as proposed. Finally, this extension gives a significant benefit of performance on deriving and processing depedency graphs which are usually very large and complex in real-life model transformations.

**Further processing of dependency graphs**

The derived dependency graphs in the study are used to identify the crosscutting behavior. However, as discussed, the effectiveness is still not very high. More research should be provided to further process these dependency graphs with respect to identify crosscutting concerns more effectively.

One reason of this ineffectiveness is that dependency graphs in a real-life model transformation are usually very large and complex, and they lead to the tendency that every element crosscuts other elements. Thus a possible processing is to reduce the complexity of dependency graphs. There are two directions which can be applied concurrently:

- *Reducing the number of elements*: elements can be considered at different degrees of granularity. For example, in a UML model, we can have a very fined degree of granularity of attributes and methods, or a coarser degree of classes, or a very coarse degree of packages. Elements at a coarser degree of granulairty contain elements at finer degrees, such as packages contain classes which in turns contain attributes and methods. Apparently, there are very few elements at a very coarse degree of granularity as compared with a very fined degree of granularity. In addition, it is observed that dealing with the problem of crosscutting concerns is more appropriate for elements at a coarser degree of granularity. Thus we can reduce the complexity of dependency graphs by keeping only the elements at a coarser degree of granularity and aggregating the mappings based on the containment property of elements.
- *Reducing the number of mappings*: a pair of a source and a target element may participate in multiple transformation rule definition and execution. It is observed that this particular pair usually involves in a few direct mappings (ideally only a single one as a target element should be derived only once from a particular source element) and in more indirect mappings (as multiple rules depend on this rule). The higher number of direct/indirect mappings of this pair means the rule is more complex and there are more rules depend on this rule; that means the mapping relation between these source and target elements is more important with respect of dealing with the problem of crosscutting concerns. Thus we can reduce the complexity of dependency graphs by counting the number of direct/indirect mappings and keeping only those pairs of source and target elements which have the number of direct/indirect mappings larger than a threshold.

The study proposed to derive the dependency graphs from the definition of the transformation rules and the tracing information of the execution of these rules. However, no explicit method was provided for this derivation. A further research could be done in order to provide a concrete algorithm to build these dependency graphs from transformation rules written in either the Relations or the Core language, and from the tracing data. The study also lacked of support of tools for transformation execution and tracing data maintenance. Future work should enhance the study in this direction.

Another issue with the dependency graphs in this study is that the crosscutting behavior is reflected not very clearly due to the fact that dependency graphs in a real-life application are usually very large and complex. The study also proposed a technique to pre-process the dependency graphs by distinguishing between direct and indirect mapping relations. Further work could be done in order to provide other techniques to fine tune the tracing model and dependency graphs so that crosscutting concerns could be identified more effectively. The above recommendations are some examples of these techniques.

# References


[1] AspectJ Team (2004). AspectJ Programming Guide. Retrieved on June 2006 from http://www.eclipse.org/aspectj

[2] Bakker, J. (2005) Traceability of Concerns. MSc Thesis, University of Twente, April 2005

[3] Berg, K.G. van den and Conejero, J. (2005). Disentangling crosscutting in AOSD: a conceptual framework, in Second Edition of European Interactive Workshop on Aspects in Software, Brussels, Belgium

[4] Berg, K.G. van den, Tekinerdogan, B. and Nguyen, H. (2006). Analysis of Crosscutting in Model Transformations, in 2nd ECMDA Traceability Workshop, Bilbao, Spain, 2006

[5] Bergmans, L. (1994). Composing Concurrent Objects, PhD Thesis, University of Twente, June 1994

[6] Borland (2006). Together Architect 2006 for Eclipse Software. Retrieved on March, 2006 from http://www.borland.com/downloads/download_together.html

[7] Czarnecki, K. and Helsen, S. (2003). Classification of Model Transformation Approaches, in Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, 2003

[8] Filman, R. E., Elrad, T., Clarke, S. and Aksit, M. (2004). Aspect-Oriented Software Development, Addison Wesley Professional, 2004

[9] Henninger, F. (2004). Analysis of Crosscutting in MDA, MSc Thesis, University of Twente, November 2004

[10] IBM Alphaworks (2006). HyperJ Website. Retrieved on June 2006 from http://www.alphaworks.ibm.com/tech/hyperj

[11] Kleppe, A., Warmer, J. and Bast, W. (2003). MDA Explained: The Model Driven Architecture™: Practice and Promise, Addison Wesley, 2003

[12] Kurtev, I. (2005). Adaptability of Model Transformation, PhD Thesis, University of Twente, 2005

[13] Masuhara, H. and Kiczales, G. (2003). Modeling Crosscutting in Aspect-Oriented Mechanisms, in ECOOP 2003, Darmstadt, Germany, 2003

[14] Nederpel, R. (2005). A QVT Model Transformation Language Represented by Graph Production Systems, MSc Thesis, University of Twente, September 2005

[15] OMG (2003). A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, OMG document ad/2003-08-02

[16] OMG (2003). Common Warehouse Metamodel™ (CWM™) Specification, v1.1, OMG document formal/2003-03-02

[17] OMG (2003). MDA Guide, version 1.0.1, OMG document omg/2003-06-01

[18] OMG (2002). Meta Object Facility Specification, version 1.4, OMG document formal/2002-04-03

[19] OMG (2003). Meta Object Facility (MOF) 2.0 Core Proposal, OMG document ad/2003-04-07

[20] OMG (2004). Metamodel and UML Profile for Java and EJB, version 1.0, OMG document formal/2004-02-02

[21] OMG (2006), Model Driven Architecture website. Retrieved on February 2006 from *http://www.omg.org/mda*

[22] OMG (2002). Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, OMG document ad/2002-04-10

[23] OMG (2005). Revised Submission for MOF 2.0 Query/Views/Transformations RFP, QVT-Merge Group, version 2.0, OMG document ad/2005-03-02

[24] OMG (2004). UML Profile for Patterns, version 1.0, OMG document formal/2004-02-04

[25] OMG (2005). Unified Modeling Language: Superstructure, version 2.0, OMG document formal/05-07-04

[26] Sutton Jr, S. M. and Rouvellou, I. (2002). Modeling of Software Concerns in Cosmos. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, pages 127--133. ACM Press, 2002

[27] TCS (2006). ModelMorf: a model transformer. Retrieved on June 2006 from http://www.tcs-trddc.com/modelmorf/index.htm

[28] TRESE (2006). Composition Filters Implementation Project. Retrieved on June 2006 from http://composestar.sf.net/

[29] TRESE (2006). GRaphs for Object-Oriented VErification (GROOVE). Retrieved on February 2006 from http://www.cs.utwente.nl/~groove

[30] TRESE (2006). The Aspect-Oriented Software Architecture Design Portal of TRESE. Retrieved on February 2006 from http://trese.cs.utwente.nl/taosad

[31] Witkop, S. (2005). DRAFT MDA User's Requirements for QVT Transformation, Version 0.23. Retrieved on February 2006 from http://www.omg.org/docs/mda-user/05-02-04.rtf

# Appendices

The appendices chapter provides complete result produced during the development of the case study. The result includes:

- The transformation definition from the PIM model to the Relational PSM model.
- The transformation definition from the PIM model to the Java PSM model.
- Dependency graphs and dependency matrices for the UML-to-Java transformation definition.

## A. UML to Relational Transformation Rules

This transformation defines transformation rules between a SimpleUML model and a SimpleRDBMS model. It contains the following relations and functions:

- `ClassToTable`: a top-level relation to maintain relationships between each class in the SimpleUML model with a table in the SimpleRDBMS model.
- `ComplexAttributeToColumn`: a top-level relation. When a persistent class in the SimpleUML model has an attribute whose type is another persistent class, this attribute is transformed to a foreign key relation between two corresponding tables in the SimpleRDBMS model.
- `AssocToFKey`: a top-level relation to maintain relationships between each association relation between two persistent classes in the SimpleUML model with a foreign key relation between two corresponding tables in the SimpleRDBMS model.
- `AttributeToColumn`: a relation to maintain relationships between each attribute in the SimpleUML model with a column in the SimpleRDBMS model. This relation is divided into 2 cases by invoking two relations `PrimitiveAttributeToColumn` and `SuperAttributeToColumn`
- `PrimitiveAttributeToColumn`: a relation to maintain relationships between each attribute of a persistent class in the SimpleUML model with a column in the SimpleRDBMS model.
- `SuperAttributeToColumn`: a relation to maintain relationships between each attribute of all super classes of a persistent class in the SimpleUML model with a column in the SimpleRDBMS model.
- `PrimitiveTypeToSqlType`: a function to convert UML primitive types to SQL types.

```
transformation umlRdbms(uml uses SimpleUML, rdbms uses SimpleRDBMS) {
key Table (name, schema);
key Column (name, owner);
key Key (name, owner);
key ForeignKey (name, owner);

top relation ClassToTable {
    cn: String;
    checkonly domain uml c:Class {
        kind='persistent',
        name=cn
    };
    enforce domain rdbms t:Table {
        schema = s:Schema {name='CFVS_PSM_RELATIONAL'},
        name = cn,
        column = cl:Column {name = 'id', type = 'NUMBER'} ,
        key = k:Key {name = cn + '_pk', column = cl}
    };
    where {
        AttributeToColumn(c, t);
    }
}

relation AttributeToColumn {
    checkonly domain uml c:Class {};
    enforce domain rdbms t:Table {};
    where {
        PrimitiveAttributeToColumn(c, t);
        SuperAttributeToColumn(c, t);
    }
}
```

```
relation PrimitiveAttributeToColumn {
    an, pn, sqltype: String;
    checkonly domain uml c:Class {
        attribute=a:Attribute {
            name=an,
            type=p:PrimitiveDataType {name=pn},
            multiplicity=multi
        }
    };
    enforce domain rdbms t:Table {
        column=cl:Column {name=an, type=sqltype }
    };
    where {
        sqltype = PrimitiveTypeToSqlType(pn, multi);
    }
}

relation SuperAttributesToColumn {
    checkonly domain uml c:Class {general=sc:Class{}};
    enforce domain rdbms t:Table {};
    where {
        AttributeToColumn(sc, t);
    }
}

top relation ComplexAttributeToColumn {
    an, fkn, scn, dcn: String;
    checkonly domain uml c:Class {
        name=scn,
        kind='persistent',
        attribute=a:Attribute {
            name=an,
            type=tc:Class {kind='persistent', name=dcn}
        }
    };
    enforce domain rdbms fkey:ForeignKey {
        name=fkn,
        owner=srcTbl,
        column=fc:Column {name=an, type='NUMBER', owner=srcTbl},
        refersTo=k:Key {name=dcn+'_pk', owner=dstTbl}
    };
    when {
        ClassToTable(c,srcTbl);
        ClassToTable(tc,dstTbl);
    }
    where {
        fkn=scn + '_' + an + '_' + dcn;
    }
}

top relation AssocToFKey {
    checkonly domain uml as:Association {
        name=asn,
        source=sc:Class {kind='persistent', name=scn},
        destination=dc:Class {kind='persistent', name=dcn}
    };
    enforce domain rdbms fk:ForeignKey {
        name=fkn,
        owner=srcTbl,
        column=fc:Column {name=fcn, type='NUMBER', owner=srcTbl},
```

```
        refersTo=k:Key {name= dcn + '_pk', owner = dstTbl}
    };
    when {
        ClassToTable(sc,srcTbl);
        ClassToTable(dc,dstTbl);
    }
    where {
        fkn = scn + '_' + asn + '_' + dcn;
        fcn = fkn + '_id';
    }
}

function PrimitiveTypeToSqlType(primitiveType:String,
                                multi:String):String {
    if (multi <> '1')
    then 'BINARY'
    else if (primitiveType = 'INTEGER')
        then 'NUMBER'
        else if (primitiveType = 'BOOLEAN')
            then 'BOOLEAN'
            else 'VARCHAR'
            endif;
        endif;
    endif;
}
}
```

***Table 14: Transformation Definition from UML Model to Relational Schema***

## B. UML to Java Transformation Rules

This transformation defines transformation rules between a SimpleUML model and a Java model. It contains the following relations and functions:

- `UInterfaceToJInterface`: a top-level relation to maintain relationships between each interface in the SimpleUML model with a Java interface in the SimpleRDBMS model.
- `UClassToJClass`: a top-level relation to maintain relationships between each class in the SimpleUML model with a Java class in the Java model.
- `AttributeToField`: a relation to maintain relationships between each attribute of a class in the SimpleUML model with a field and a pair of set/get methods of the corresponding Java class in the Java model.
- `OperationToMethod`: a relation to maintain relationships between each operation of a class in the SimpleUML model with a method of the corresponding Java class in the Java model.
- `UParameterToJParameter`: a relation to maintain relationships between each parameter of a method in the SimpleUML model with a Java parameter of the corresponding method in the Java model.
- `AssociationToField1`, `AssociationToField2`: top-level relations to transform association relations between two classes in the SimpleUML model to fields and set/get methods in the corresponding Java classes in the Java model.
- `ImplementationToImplements`: a top-level relation. When a class in the SimpleUML model has an implementation relation with another class, the `implements` property of the Java class corresponding to the former UML class should include the Java class corresponding to the latter UML class.
- `GeneralizationToExtends`: a top-level relation. When a class in the SimpleUML model has a generalization relation with another class, the `extends` property of the Java class corresponding to the former UML class should be set to the Java class corresponding to the latter UML class.
- `MessageToImports`: a top-level relation to maintain relationships between each message calls of sequence diagrams in the SimpleUML model with the `imports` property of the corresponding Java class in the Java model.

```
transformation Uml2Java (uml uses SimpleUML, java uses JavaMetamodel)
{
    key JavaClass (name);
    key Field (name, owner);
    key Method (name, owner);
    key JavaParameter (name, owner);

    top relation UInterfaceToJInterface {
        uin: String;
        checkonly domain uml ui:Class {
            stereotype="<<interface>>"
        };
        enforce domain java ji:JavaClass {
            stereotype="<<interface>>"
        };
        when {
            UClassToJClass(ui, ji);
        }
    }

    top relation UClassToJClass {
        ucn: String;
        checkonly domain uml uc:Class {name=ucn};
```

```
        enforce domain java jc:JavaClass {name=ucn};
        where {
            AttributeToField (uc, jc);
            OperationToMethod (uc, jc);
        }
    }

    relation AttributeToField {
        an, umltype, javatype: String
        checkonly domain uml uc:Class {
            attribute=a:Attribute {
                name=an,
                type=ut:Classifier{name=umltype}
            }
        };
        enforce domain java jc:JavaClass {
            fields=f:Field {name='m_'+an, type=javatype},
            methods=get:Method {
                name='get'+an,
                returnParameter=jp1:JavaParameter {type=javatype}
            },
            methods=set:Method {
                name='set'+an,
                returnParameter=jp2:JavaParameter {type='void'},
                inputParameters=jp3:JavaParameter {
                    name=an, type=javatype
                }
            }
        };
        where {
            javatype=UTypeToJType(umltype);
        }
    }

    relation OperationToMethod {
        opn, umltype, javatype: String;
        checkonly domain uml uc:Class {
            operation = op:Operation {
                name = opn,
                returnType = ut:Classifier {name=umltype}
            }
        };
        enforce domain java jc:JavaClass {
            methods = mt:Method {
                name=opn,
                returnParameter = jp:JavaParameter {type = javatype}
            }
        };
        where {
            javatype = UTypeToJType(umltype);
            UParameterToJParameter(op, mt);
        }
    }

    relation UParameterToJParameter {
        upn, umltype, javatype: String;
        checkonly domain uml op:Operation {
            parameter = up:Parameter {
                name = upn,
                direction != 'return',
                type = ut:Classifier {name = umltype}
```

```
            }
        };
        enforce domain java mt:Method {
            inputParameters = jp:JavaParameter {
                name = upn,
                type = javatype
            }
        };
        where {
            javatype = UTypeToJType(umltype);
        }
    }

    top relation AssociationToField1 {
        srcUcn, srcRole, srcMult, ascType: String;
        checkonly domain uml asc:Association {
            destination = dstUc:Class {},
            source = srcUc:Class {name = srcUcn},
            sourceRole = srcRole,
            sourceNavigability != 'NOT_NAVIGABLE',
            sourceMultiplicity = srcMult
        };
        enforce domain java jc:JavaClass {
            fields = f:Field {name = 'm_' + srcRole, type = ascType},
            methods = get:Method {
                name = 'get' + srcRole,
                returnParameter = jp1:JavaParameter {type = ascType}
            },
            methods = set:Method {
                name = 'set' + srcRole,
                returnParameter = jp2:JavaParameter {type = 'void'}
                inputParameters = jp3:JavaParameter {
                    name = srcRole, type = ascType
                }
            }
        };
        when {
            UmlClassToJavaClass(destUc, jc);
        }
        where {
            ascType = GetAscType(srcUcn, srcMult);
        }
    }

    top Relation AssociationToField2 {
        dstUcn, dstRole, dstMult, ascType: String;
        checkonly domain uml asc:Association {
            source = srcUc:Class {},
            destination = dstUc:Class {name = dstUcn},
            destinationRole = dstRole,
            destinationNavigability != 'NOT_NAVIGABLE',
            destinationMultiplicity = dstMult
        };
        enforce domain java jc:JavaClass {
            fields = f:Field {name = 'm_' + dstRole, type = ascType},
            methods = get:Method {
                name = 'get' + dstRole,
                returnParameter = jp1:JavaParameter {type = ascType}
            },
            methods = set:Method {
                name = 'set' + dstRole,
```

```
                returnParameter = jp2:JavaParameter {type = 'void'},
                inputParameters = jp3:JavaParameter {
                    name = dstRole,
                    type = ascType
                }
            }
        };
        when {
            UmlClassToJavaClass(srcUc, jc);
        }
        where {
            ascType = GetAscType(dstUcn, dstMult);
        }
    }

    top Relation ImplementationToImplements {
        cliUcn, supUin: String;
        checkonly domain uml imp:Implementation {
            clientNamedElement = cliUc:Class {name=cliUcn},
            supplierNamedElement = supUi:Interface {name=supUin}
        };
        enforce domain java jc:JavaClass {
            name = cliUcn,
            implements = ji:JavaInterface {name=supUin}
        };
        when {
            UmlClassToJavaClass
        }
    }

    top Relation GeneralizationToExtends {
        cliUcn, supUcn: String
        checkonly domain uml gen:Generalization {
            clientNamedElement = cliUc:Class {name=cliUcn},
            supplierNamedElement = supUc:Class {name=supUcn}
        };
        enforce domain java jc1:JavaClass {
            name=cliUcn,
            extends = jc2:JavaClass {name=supUcn}
        };
    }

    top Relation MessageToImports {
        checkonly domain uml msg:Message {
            sendEvent = sendClass:Class {},
            receiveEvent = recvClass:Class {}
        };
        enforce domain java sendJClass:JavaClass {
            importsClass = recvJClass:JavaClass {}
        };
        when {
            UmlClassToJavaClass(sendClass, sendJClass);
            UmlClassToJavaClass(recvClass, recvJClass);
        }
    }
}
```

***Table 15: Transformation Definition from UML Model to Java***

## C. Dependency graphs for UML To Java Transformation Rules

This appendix presents the dependency graphs and dependency matrices which are derived for each transformation rule of the UML-To-Java transformation.

Dependency graphs at both metamodel and model levels are derived. The dependency graph at the metamodel level shows the mapping between types of elements in a transformation rule. The dependency graph at the model level shows the mapping between elements of the SimpleUML and Java model due to the execution of this rule; i.e. it shows presents instances of the tracing class corresponding to this rule.

There are dependency matrices at metamodel and model levels as well. In the dependency matrix at the metamodel level, the header column lists the types of elements of the source model and the header row lists the types of elements of the target model. In the dependency matrix at the model level, the header column lists the elements of the source model and the header row lists the elements of the target model. For both types of dependency matrices, an inner cell with a value 1 indicates a mapping between the corresponding source and target (types of) elements. A light gray row represents a scattering case, a light gray column represents a tangling case, and dark gray cell (an intersect of a light gray row and a light gray column) represents a crosscutting point.

**Rule: UClassToJClass**

This transformation rule transforms UML classes to Java classes. The dependency graph at the metamodel level shown in Figure 48 contains only a single direct mapping between a Class and a JavaClass element.

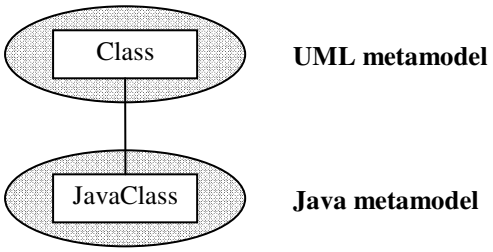



***Figure 48: Dependency graph (metamodel) for UClassToJClass***

| target<br>source | JavaClass |
|---|---|
| UMLClass | 1 |

***Table 16: Dependency matrix (metamodel) for UClassToJClass***

The dependency graph at the model level shown in Figure 49 contains direct mappings Product-Product, Branch-Branch, Security-Security, etc.

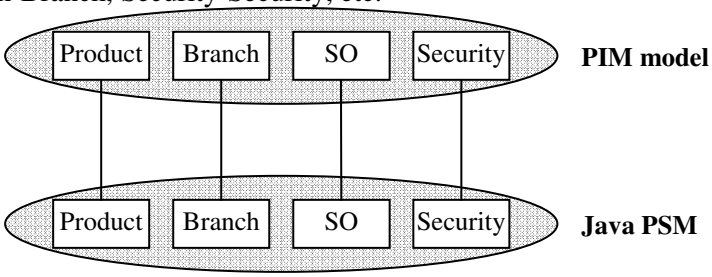



***Figure 49: Dependency graph (model) for UClassToJClass***

| target<br>source | Product | Branch | SecuredObject | Security |
|---|---|---|---|---|
| Product | 1 | | | |
| Branch | | 1 | | |
| SecuredObject | | | 1 | |
| Security | | | | 1 |

***Table 17: Dependency matrix (model) for UClassToJClass***

**Rule: AttributeToField**

Note: More precisely, the dependency data here is for the rule AttributeToField_UClassToJClass rule which is derived from the rule AttributeToField when it is invoked from the rule UClassToJClass. The indirect mapping shown in the dependency graph and its elements are passed in by the rule UClassToJClass.

Apparently, this rule is not good because it contains the crosscutting relation. This crosscutting relation is more obvious in the dependency graph at the model level. Ideally, the source metamodel element Classifier should be transformed to corresponding target metamodel elements in a separate rule.

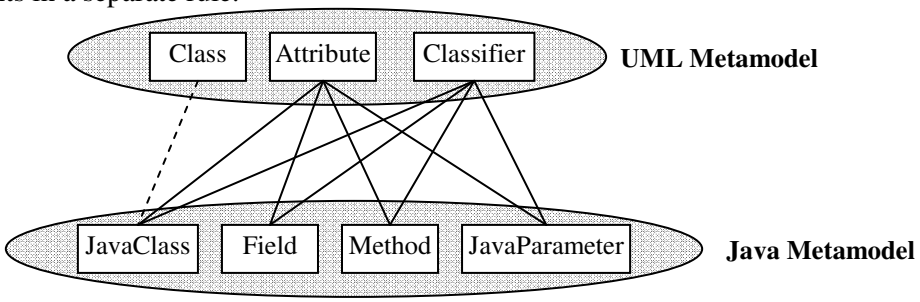



***Figure 50: Dependency graph (metamodel) for AttributeToField***

| target / source | JavaClass | Field | Method | JavaParameter |
|---|---|---|---|---|
| Class | 1 | | | |
| Attribute | 1 | 1 | 1 | 1 |
| Classifier | 1 | 1 | 1 | 1 |

***Table 18: Dependency matrix (metamodel) for AttributeToField***

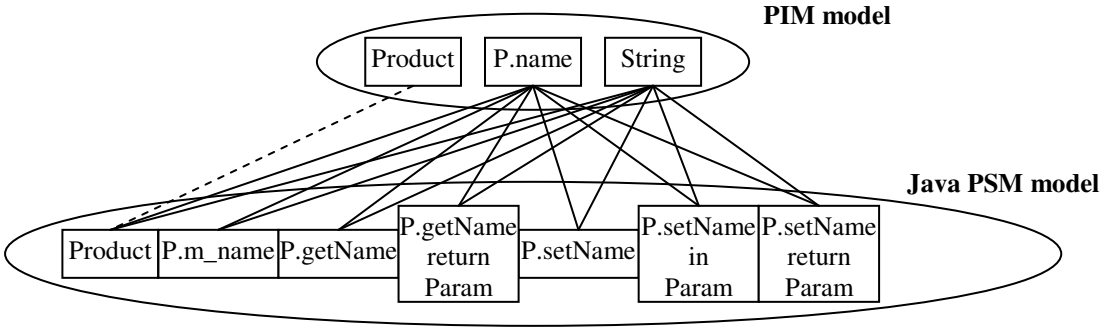



***Figure 51: Dependency graph (model) for AttributeToField***

| target / source | Product | Product. m_name | Product. getName | P.getName return Param | Product. setName | P.setName in Param | P.setName return Param |
|---|---|---|---|---|---|---|---|
| Product | 1 | | | | | | |
| P.name | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| String | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

***Table 19: Dependency matrix (model) for AttributeToField***

**Rule: OperationToMethod**

This rule has the same problem as the rule **AttributeToField** in which its dependency graph contains crosscutting relations. Thus the rule should also be rewritten to eliminate this relation.

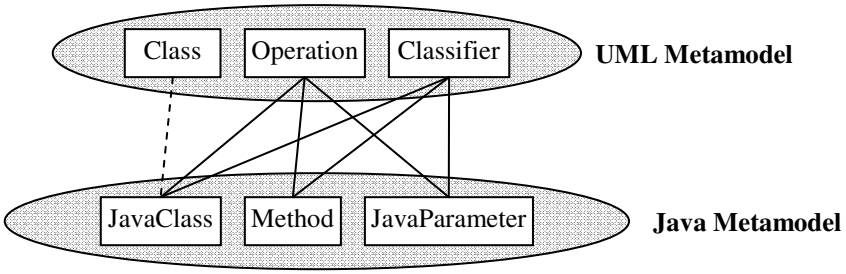



***Figure 52: Dependency graph (metamodel) for OperationToMethod***

| target / source | JavaClass | Method | JavaParameter |
|---|---|---|---|
| Class | 1 | | |
| Operation | 1 | 1 | 1 |
| Classifier | 1 | 1 | 1 |

***Table 20: Dependency matrix (metamodel) for OperationToMethod***

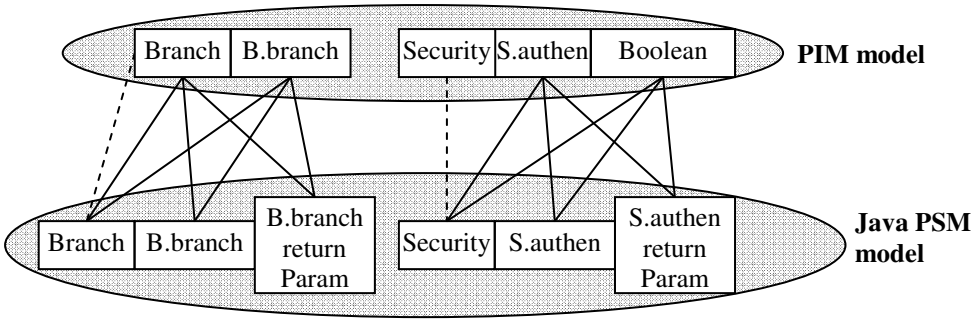



***Figure 53: Dependency graph (model) for OperationToMethod***

| target / source | Branch | B.branch | B.branch return Param | Security | S.authenticate | S.authenticate return Param |
|---|---|---|---|---|---|---|
| Branch | 1 | 1 | 1 | | | |
| B.branch | 1 | 1 | 1 | | | |
| Security | | | | 1 | | |
| S.authenticate | | | | 1 | 1 | 1 |
| Boolean | | | | 1 | 1 | 1 |

***Table 21: Dependency matrix (model) for OperationToMethod***

**Rule: MessageToImports**

The dependency graphs for this rule at both metamodel and model levels, combined with the dependency graphs for the rule UClassToJClass in Figure 48 and Figure 49, show that the `Security` class crosscuts other elements of the PIM model, such as `Product`, `Branch` and message calls from `Product` to `Security` in the `add` method of the `Product` class.

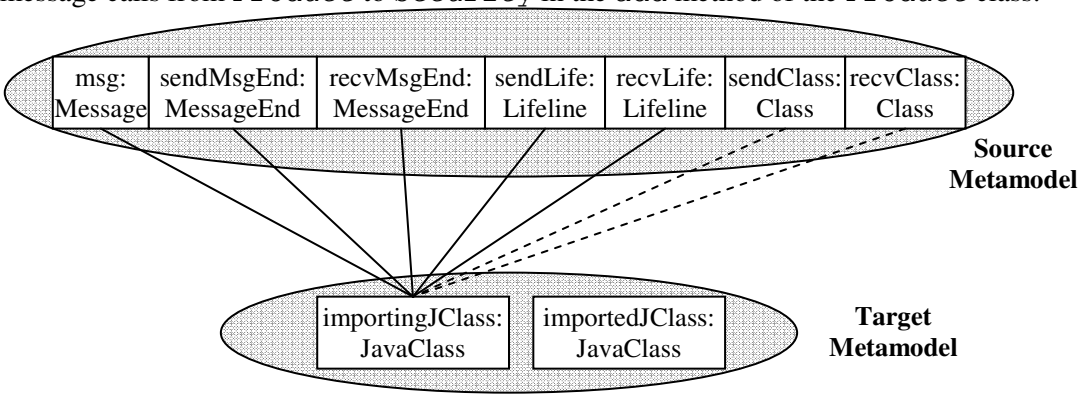
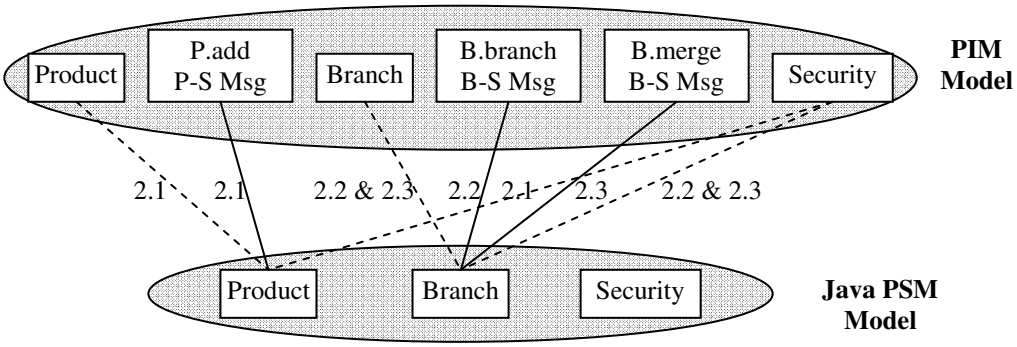



***Figure 54: Dependency graph (metamodel) for MessageToImports***

| target / source | JavaClass |
|---|---|
| Class | 1 |
| Message | 1 |

***Table 22: Dependency matrix (metamodel) for MessageToImports***




**Note:** Some elements are omitted for clearer view

***Figure 55: Dependency graph (model) for MessageToImports***

| target / source | Product | Branch | Security |
|---|---|---|---|
| Product | 1 | 0 | 0 |
| P.add P-S Msg | 1 | 0 | 0 |
| Branch | 0 | 1 | 0 |
| B.branch B-S Msg | 0 | 1 | 0 |
| B.merge B-S Msg | 0 | 1 | 0 |
| Security | 1 | 1 | 0 |

***Table 23: Dependency matrix (model) for MessageToImports***

***Abbreviation:***

*P.add P-S Msg*: the Message element from a Product object to a Security object in the sequence diagram for the operation Product.add

*B.branch B-S Msg*: the Message element from a Branch object to a Security object in the sequence diagram for the operation Branch.branch

*B.merge B-S Msg*: The Message element from a Branch object to a Security object in the sequence diagram for the operation Branch.merge