UNIVERSIDAD DE LOS ANDES
FACULTY OF ENGINEERING
Department of
Systems and Computer Engineering
Software Construction Group

VRIJE UNIVERSITEIT BRUSSEL
FACULTY OF SCIENCE
Department of
Computer Science
Software Languages Lab

A non-invasive approach for evolving Model Transformation Chains

---

Andrés Yie

February 2011

A dissertation submitted in partial fulfillment
of the requirements for the degrees
of Doctor of Science at Vrije Universiteit Brussel
and Doctor of Engineering at Universidad de los Andes

Promotors:
Prof. Dr. Rubby Casallas
Prof. Dr. Viviane Jonckers
Dr. Dirk Deridder

Jury:
Prof. Dr. María Cecilia Bastarrica
Prof. Dr. Alfonso Pierantonio
Prof. Dr. Jorge Villalobos
Prof. Dr. Dario Correal
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Beat Signer

Dedication

Al chinito y su burbujita, y por supuesto a mi Adri.
(To my parents, and of course, to my wife.)

Abstract

Model-Driven Engineering (MDE) approaches promote the use of models expressed in terms of problem domain concepts (e.g. Entities, Services, etc) as prime artifacts to develop software. These models, to which we refer as high-level models, can be used as input for a *Model Transformation Chain* (MTC). This chain is a sequence of transformation steps that converts the high-level model, which is rooted in the problem domain, into a low-level model that is rooted in the solution domain (e.g., Java, C#). In addition to the translation from problem domain concepts to solution domain concepts (e.g., Java Class, Java Annotation), the transformation chain adds implementation details in every transformation step. In the last step of the chain, there is a model-to-text transformation that produces the code of the software system.

Evolution is an inherent characteristic of software systems. For instance, a software system must evolve if it is required to include new functionality, new non-functional properties, or the migration of the technology platform. Similar to this, MTCs are also susceptible to evolution. The evolution of an MTC confronts us with several problems mainly related to the strong dependencies between metamodels and models, metamodels and transformations, and between each transformation step and the following.

The particular problem we address is the addition of a new concern (e.g., security, monitoring) that was not anticipated in an existing MDE implementation. No real problem arises if the new concern can be cleanly expressed using the existing high-level metamodel. However, if this is not the case, the addition of the new concern will lead to a number of maintenance problems, such as metamodels that are polluted with alien concepts, models which no longer conform to the metamodels, transformations that are no longer compatible with the new metamodels, or broken dependencies between transformations. All of these problems are a detriment to the evolution of an existing MTC and a single change can cause a ripple effect through every transformation step.

We present a strategy that consists of modularizing the new concern in a

new concern-specific MTC. First, we specify the new concern in a separate high-level model. This leaves the original model unaltered and unaware of the added concern. The concern-specific model can thus be specified using concepts close to its domain which are expressed in a separate metamodel. Therefore, we have two high-level models that conform to two different metamodels. We align the high-level models using a *Correspondence Model* (CM) [BBDF+06], which explicitly describes correspondence relationships among the elements of different models. Next, both models are transformed by the MTCs (i.e., the existing MTC and the new concern-specific MTC) that produce two complementary low-level models that conform to the same metamodel. We use the CM between the high-level models to derive a new CM between the lowest level models. We use these correspondence relationships to identify the elements to compose. Finally, we perform a composition between homogeneous concepts that conform to the same metamodel. Furthermore, when the lowest level models conform to different metamodels (i.e., different technological platforms) it is possible to use the correspondence relationships to check consistency between the models or to generate code-level composition specifications (e.g., the XML descriptors between a Java application and a database).

We have developed a mechanism to automatically derive the CM through the various steps in the MTCs. This mechanism allows us to reuse the presented approach to add additional concern-specific MTCs to an existing MTC reducing the impact to the existing MTC artifacts and modularizing the changes.

In this dissertation we present an approach that reduces the complexity of evolving a model transformation chain. Our approach offers several advantages: 1) it reuses the existing assets (metamodels, models and transformations), 2) it modularizes the changes in a new set of metamodels, models and transformations, 3) it facilitates the modeling of different concerns in separated models and close to the problem domain, 4) it offers an automatic derivation mechanism to identify the elements to compose in the low-level models based on relationships defined in the high-level, and 5) it eases the use of a reusable mechanism to integrate the changes.

Samenvatting

Model-Driven Engineering (*MDE*) methodes stimuleren het gebruik van modellen uitgedrukt in concepten behorende tot het probleemdomein (zoals Entities, Services, etc) als primaire artefacten om software te gaan ontwikkelen. Deze modellen, waarnaar gerefereerd wordt als *hoog niveau modellen*, kunnen gebruikt worden als input voor een *Model Transformation Chain* (MTC). Een MTC stelt een opeenvolging van tranformatiestappen voor die een hoog niveau model, behorende tot het probleemdomein, omzet in een laag niveau model, behorende tot het oplossingsdomein (e.g. Java, C#). Aanvullend aan de omzetting van probleemdomein concepten naar oplossingsdomein concepten (e.g. Java Class, Java Annotation) voegt de transformatie chain implementatiedetails toe aan elke transformatiestap. In de laatste stap van de chain vindt er een *model-to-text* transformatie plaats die dan de code van het software systeem produceert.

Evolutie is een inherent kenmerk van software systemen. Een software systeem moet bijvoorbeeld evolueren als nieuwe functionaliteiten of nieuwe niet-functionele eigenschappen moeten worden toegevoegd of als het technologisch platform gemigreerd moet worden. Gelijkaardig hieraan is dat MTCs ook gevoelig zijn voor evolutie. De evolutie van een MTC brengt verschillende problemen met zich mee hoofdzakelijk te wijten aan de sterke afhankelijkheden tussen metamodellen en modellen, metamodellen en transformaties, en tussen elke transformatiestap en de daaropvolgende.

Het specifieke probleem dat we hier willen aanpakken is de toevoeging van een nieuw concern (zoals bv. security, monitoring) aan een bestaande implementatie die niet voorzien was in een reeds bestaande MDE implementatie. Dit levert geen echte problemen op als de nieuwe concern netjes kan worden uitgedrukt in het bestaande hoog niveau metamodel. Indien dit echter niet het geval is zal het toevoegen van deze nieuwe concern leiden tot een aantal onderhoudsproblemen, zoals metamodellen die vervuild zijn met vreemde concepten, modellen die niet langer conform zijn met de metamodellen, transformaties die

niet meer compatibel zijn met de nieuwe metamodellen, ofwel gebroken links tussen verschillende transformaties. Al deze problemen doen afbraak aan de evolutie van een bestaande MTC en één enkele verandering kan een cascade effect teweegbrengen doorheen elke transformatiestap.

In deze thesis stellen we een strategie voor waarbij het nieuwe concern gemodularizeerd wordt in een nieuwe concern-specifieke MTC. Ten eerste gaan we het nieuwe concern specificeren in een afzonderlijk hoog niveau model zodanig dat het oorspronkelijke model onveranderd blijft en ook niets afweet van het toegevoegde concern. Het concern-specifieke model kan bijgevolg gedefinieerd worden in domein-specifieke concepten die worden uitgedrukt in een afzonderlijk metamodel. Zo hebben we dus twee hoog niveau modellen die conform zijn met twee verschillende metamodellen. De hoog niveau modellen worden met elkaar gelinkt aan de hand van een *Correspondentie Model* (CM) [BBDF+06], dat expliciet de correspondentie relaties omschrijft tussen de elementen van de verschillende modellen. Vervolgens worden beide modellen getransformeerd door de MTC's (i.e. de bestaande MTC en de nieuwe concern-specifieke MTC) met als resultaat twee complementaire laag niveau modellen die conform zijn met hetzelfde metamodel. We gebruiken het correspondentiemodel tussen de hoog niveau modellen om een nieuw correspondentiemodel af te leiden tussen de laag niveau modellen. We gebruiken de correspondentie relaties om de samen te stellen elementen te identificeren. Tenslotte maken we een samenstelling van de homogene concepten die conform zijn met hetzelfde metamodel. Wanneer de laag niveau modellen conform zijn met verschillende metamodellen (i.e. verschillende technologische platformen) is het mogelijk om de correspondentie relaties te gebruiken om te checken of de modellen nog steeds consistent zijn ten opzichte van elkaar, ofwel om compositie specificaties op code niveau te gaan genereren (e.g. de XML descriptors tussen een Java applicatie en een databank).

We hebben een mechanisme ontworpen om automatisch het CM te kunnen afleiden doorheen de verschillende stappen in de MTC. Dit mechanisme laat ons toe om de hier voorgestelde aanpak te gaan hergebruiken om additionele concern-specifieke MTC's te gaan toevoegen aan een bestaande MTC, daarbij de impact reducerend voor de bestaande MTC artefacten alsook het modularizeren van de aanpassingen.

In deze doctoraatsthesis stellen we een aanpak voor die de complexiteit reduceert voor het evolueren van een Model Transformation Chain. Onze aanpak biedt de volgende voordelen: 1) hergebruik van bestaande elementen (modellen, metamodellen en transformaties) is mogelijk, 2) aanpassingen kunnen gemodulariseerd worden in een nieuwe set van metamodellen, modellen en transformaties, 3) modellering van verschillende concerns in aparte modellen wordt vergemakkelijkt en laat toe om dicht bij het probleemdomein te blijven, 4) een automatisch afleidingsmechanisme wordt aangeboden om de elementen te identificeren om de low-level modellen te kunnen samenstellen

gebaseerd op relaties gedefinieerd op een hoog abstractieniveau, en 5) gebruik van een herbruikbaar mechanisme om de veranderingen te gaan integreren wordt makkelijker gemaakt.

Acknowledgements

In this long and challenging journey, I received a large amount of support, guidance and friendship. This is the place to show my gratitude to all the people who walked with me and to make my Ph.D. possible.

First of all, I would like to thank my promoter Rubby Casallas not only for offering me the opportunity to pursue this degree and accepting me as a PhD student, but for teaching and guiding me for many years. I'd like to thank her for her confidence, support and hard work throughout all these years. I hope to reward her confidence by concluding my Ph.D. work with this dissertation.

The next person to whom I am especially grateful is my promoter Dirk Deridder. He helped me to maintain the motivation in the difficult times and the multiple discussions with him and his ever-accurate feedback improved my work and kept my enthusiasm for doing research. I truly thank him for all the effort that he spent on reading in detail my dissertation document.

Thank you to my promoter Viviane Jonckers for her support throughout all these years. As head of the System and Software Engineering Lab where I spent almost half of my research years, she succeeded in creating a challenging environment for research.

I would like to thank the members of my jury committee, Jorge Villalobos, Dario Correal, Beat Signer, Wolfgang De Meuter, María Cecilia Bastarrica, and Alfonso Pierantonio for taking the time to read this dissertation in detail and for providing me with valuable feedback.

Thank you to the 'Vlaamse Interuniversitaire Raad (VLIR)' as they funded the CARAMELOS project and the Universidad de Los Andes for providing me with the financial support to carry out my research.

A special thank you goes to Oscar Gonzalez and Mario Sanchez for being my partners in the Ph.D. path. The three of us lived this process as a group, supporting each other, learning from one another and growing together. It was nice to 'walk' with you guys. Additionally, I want to say thank you to Gigi and Diana for understanding and supporting them.

I am grateful to the members of the QualDev research group for their collaboration in my work. In particular, many thanks go to my colleagues and friends, Hugo Arboleda, Rafael Meneses and Yeimi Peña with whom I have shared good experiences and I have had interesting discussions.

I thank my colleagues at Software Languages Lab for the stimulating research environment. I thank the friendship from my colleagues at the former SSEL lab: Bruno De Fraine, Dennis Wagelaar, Niels Joncheere, Ragnhild Van Der Straeten, and Eline Philips. I particularly want to thank Dennis for the interesting discussions, guidance and help in the process of understanding the dirty stuff inside the ATL-VM. Additionally, I want to say thank you to my good friends Jorge Vallejos and Isabel Michiels with whom I have had deep and motivating conversations about the Ph.D., life and so many things. I especially want to thank Isabel for helping me with the Dutch version of the abstract

Being so far away from home was not always easy. I would like to thank all my friends in Brussels, who helped me feel more at home and who accompanied me during the years that I spent in Belgium. I want to say thank you to Carlos Alvarez and Adriana Sotelo for taking care of me and becoming my family. Rob Vanmeert for talking, playing, partying and being one of my best friends. Carlos Noguera and Angela Lozano who shared with me so many Belgian days. Sebastian Gonzalez, Nicolas Cardozo, Frank van der Kleij, Sonia Petitprez, Nanny De Roover, Erik Hendrix, Lynsey Fox, Hernan Rios and Nadine Zillich for being really good friends and sharing so many good things.

I want to say thank you to Giovanna Aguirre and Sarah Faur for distracting me from the Ph.D. work, giving me some air to continue with my work. I especially want to thank Sarah for reading, reviewing and correcting my dissertation text. Additionally, I want to thank Changhee Hahn and Lina Saldarriaga for their last-minute help with my text. Thank you to Juan Bohorquez as well for starting with me in the research path, he is a really good friend and I hope to work with him again in the future.

I would like to thank to my wife's family who has welcomed me as part of their family and has helped me in the Ph.D. and so many other things. Many thanks to Pacho, Teresita, Andres, Nata and Franco. Especially I want to say thank you to Franco for helping me proofread my papers and dissertation.

I thank my family, who taught me to love knowledge. My dad who has been always with me, my mom being my heart and soul, my brothers and sisters which had been my strength, and helped me to take care of my family when I was working on my Ph.D.

Finally, I want to thank my wife Adriana. She has been my friend, my partner, and my lover during all these years. She quit her job and crossed the ocean to be with me in Belgium. Then she quit her dream job again to come back with me to Colombia. She has been taking care of me, of the house and of my family, always with her strong and incredible smile and positivism that can move mountains. From the bottom of my heart, thank you.

Table of Contents

List of Figures

List of Tables

Abbreviations

AMW Atlas Model Weaver

AOM Aspect Oriented Modeling

AOP Aspect Oriented Programming

AOSD Aspect Oriented Software Development

AS Application Server

ATL Atlas Modeling Language

ATL-VM ATL Virtual Machine

CDM Correspondence Derivation Model

CDMM Correspondence Derivation Metamodel

CIM Computation Independent Model

CM Correspondence Model

CMM Correspondence Metamodel

CMT Correspondence Model Transformation

CRUD Create, Read, Update and Delete

DSL Domain Specific Language

DSML Domain Specific Modeling Language

EA Enterprise Applications

EMF Eclipse Modeling Framework

GPL General Purpose Language

HOT High Order Transformation

JEE Java Enterprise Edition

JSF JavaServer Faces

MDA Model-Driven Architecture

MDE Model-Driven Engineering

MOF Meta-Object Facility

MTC Model Transformation Chain

MTC Model Transformation Chain

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query/View/Transformation

RBAC Role Based Access Control

SoC Separation of Concerns

TM Tracing Model

UML Unified Modeling Language

Chapter 1

Introduction

1.1 Problem Statement

Model-Driven Engineering (MDE) [Sch06] is a software development approach that uses models as a mechanism to increase the level of abstraction. MDE uses models as first class entities in the development process, meaning that models are main artifacts throughout the entire development process. This is a fundamental difference with other approaches (e.g. code-centric approaches) that use models only as means of representation, documentation, and communication.

A large number of MDE-based approaches promote the use of models expressed in terms of problem domain concepts (e.g. Entities, Services) as the prime artifact to develop software. These models, to which we refer as high-level models, are used as input for a *Model Transformation Chain* (MTC). An MTC is a sequence of transformation steps that converts the high-level model, which is rooted in the problem domain, into a low-level model that is rooted in the solution domain (e.g., Java, C#). In addition to the translation from problem domain concepts to solution domain concepts (e.g., Java Class, Java Annotation), the transformation chain adds implementation details in every transformation step. The last step of the chain is typically a model-to-text transformation that produces the code of the software system.

MDE implementations are an improvement to software development practices, in terms of increasing the abstraction by specifying applications close to the problem domain and using reusable transformations to generate their platform specific implementations. However, one is confronted with a number of problems if an MDE implementation needs to evolve. Evolution is an inherent characteristic that affects software systems, for instance, to include new functionality, to address new non-functional properties, or to migrate to a new technological platform. The evolution of an MTC confronts us with several problems, which are mainly related to the strong coupling that exists

between metamodels and models, metamodels and transformations, and each transformation step and those which follow it.

The particular problem we address in this dissertation is the addition of a new concern such as security or monitoring in an existing MDE implementation. No real problem arises if the new concern can be *cleanly*[1] expressed using the existing high-level metamodel. However, if this is not the case, the addition of the new concern will lead to a number of maintenance problems, such as metamodels that are polluted with alien concepts, models which no longer conform to the metamodels, transformations that are no longer compatible with the new metamodels, or broken dependencies between transformations. All of these problems are a detriment to the evolution of an existing MTC and a single change can cause a *ripple effect* through every transformation step.

In order to illustrate the problems that arise when a new concern is added, we will use an implementation of an existing MTC. This MTC was developed to produce Enterprise Applications (EA) [SSJ02]. An EA is a component-based application that supports the complex business requirements of an enterprise. EAs are Web-based and must satisfy complex non-functional requirements (i.e., quality attributes) such as scalability, performance, security, etc. Usually, EAs run in an Application Server (AS) that provides some support services such as load balancing, security, etc. The most widely used AS are .NET and Java Application Servers (e.g., Glassfish, JBoss), which implement the Java Enterprise Edition (JEE) specification [CS09]. Our example MTC generates EAs that run in a Java Application Server.

Using our example MTC we can generate JEE applications that run in a Java AS. Some of the examples that we have developed using the MTC are: a project management application, a change management application, a risk management application, etc. The common functionality in all these applications is managing the information of their entities and their relationships. The information of each entity is accessed and modified through basic *Create, Read, Update,* and *Delete* (CRUD) services.

For instance, suppose that we need to specify a risk management application named `Risk`. This application needs to manage multiple software projects that are developed in a company and the risks that affect each project (e.g., personnel shortage, missing requirements, schedule slips). A project is threatened by a set of risks and each risk has associated some mitigation plans. These plans allow the reduction of the impact of each threat in the project.

In order to generate a JEE application using the example MTC, it must be specified by the business expert in a high-level model using concepts that belong to a *Business metamodel*. This means that the application is specified in terms of business entities, their attributes, their relationships and the services that they provide. Next, the high-level model is transformed by the MTC

[1] *Cleanly* means not forcing the use of existing concepts to express an alien concern

into a low-level model that conforms to a *Java metamodel*. The MTC translates the high-level concepts into Java concepts. Additionally, the MTC adds implementation details that are required to execute the application. Finally, the low-level model is transformed into Java code with all the necessary information to be executed in an AS. Figure 1.1 shows the example MTC, which produces the JEE application from a high-level input model. The MTC uses a high-level model that conforms to a *Business metamodel* and transforms it into a low-level model that conforms to a *Java metamodel*. Finally, this model is transformed into Java code.

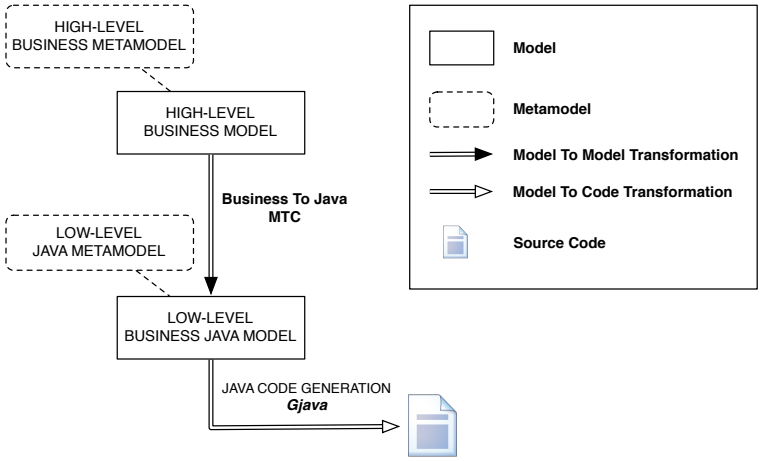


Figure 1.1: An instance of a Model Transformation Chain (MTC)

Using the Business metamodel concepts, a business expert can specify a high-level model that represents a desired application. In our example, in order to specify a risk management application, the business expert must model the risk management *business*. In other words, the business expert must specify all risk management *entities*, their *attributes*, their *relationships* and the *services* that they provide. For instance, the goal of a risk management application is to identify the risks that threaten a project. Hence, the business expert must specify the projects, the risks, and the relationship between them in the high-level model.

Figure 1.2 presents a fragment of the *Business metamodel* used by the example MTC. The goal of this metamodel is to specify EAs using high-level business concepts that are free of technological platform details. The focus of this metamodel is to describe the business, the entities that belong to it, the relationships between them, and the services that each one offers. The main concepts of this metamodel are: *Business* representing the business to describe (e.g., project management, risk management), *BusinessEntity* representing the different business entities that belong to the business (e.g., project, risk), *Attribute* representing the attributes of a *BusinessEntity* (e.g., the name

of a project), *Association* representing the relationships among the different *BusinessEntities* (e.g., the relationship between a project and the risks that threaten it), and *Service* representing the services that each *BusinessEntity* offers (e.g., the services that allow to add new risk to a project).

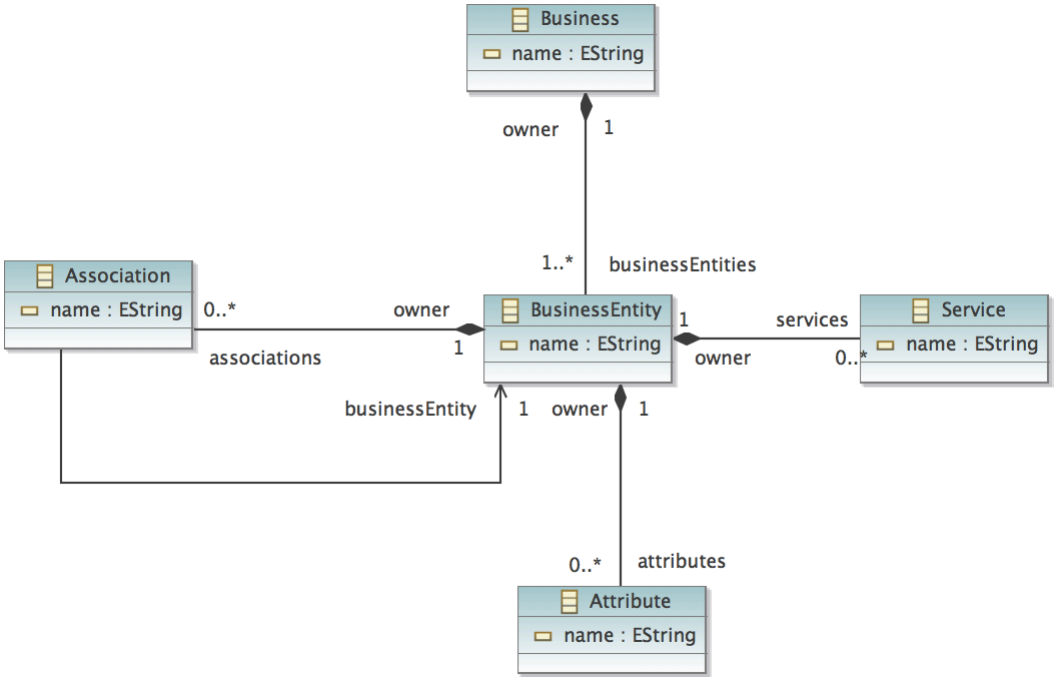


Figure 1.2: Business metamodel

In the `Risk` high-level model, we will use the concept *BusinessEntity* to describe each one of the entities that belong to the risk management business. Figure 1.3 shows a fragment of a *Business model*[2] for `Risk` that conforms to the *Business metamodel*. This model presents two *BusinessEntities* named `Project` and `Risk`, their *Attributes* (e.g., the project `name`), and an *Association* between both *BusinessEntities*, which represents the risks that belong to a project, called `risks`.

The *Business model* is processed through a series of transformation steps, and finally a JEE application is produced. The generated application offers basic JEE data management CRUD services of the business entities specified in the high-level model. However, the generated application does not support any authorization mechanism to protect access to the data and the different services offered by it. The evolution with which we will illustrate the MTC problems is the introduction of requirements for authorization support.

Having in mind to illustrate the addition of a new concern, we will evolve

[2]In this dissertation we will use the UML Object Diagram's concrete syntax to represent models.

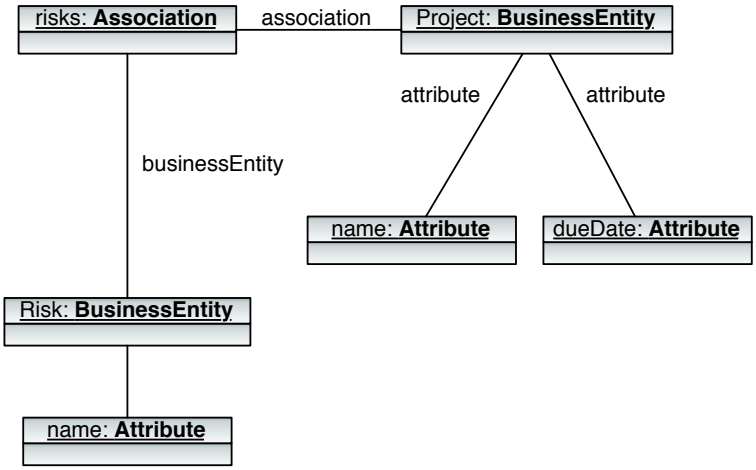


Figure 1.3: Business model

the presented MTC to generate secure JEE applications. In the evolution of the MTC we want to follow the MDE vision also used in the original MTC by specifying authorization policies at a high-level of abstraction. Adding the authorization policies at a high-level offers two major benefits: 1) it allows to define the authorization policies using concepts close to the security domain thus giving an appropriate level of abstraction to the security experts, 2) it allows to use the generative power of the MDE implementation in order to obtain the required application with the platform specific mechanisms for authorization support (i.e., JEE annotations).

In the following sections we present the problems that are originated by the addition of a new concern to an existing MTC. We need to avoid these problems in order to reduce the impact of change in the evolution of an MTC. These problems are: 1) the *detriment of maintainability and usability of the MTC artifacts* caused by directly modifying them in order to support a new concern, 2) *metamodel, model and transformations co-evolution* when the metamodel is adapted and the conformance relationship between metamodels and models is broken, and the transformations that use it are no longer compatible, 3) *ripple effect* due to the strong dependencies that exist between the MTC artifacts, a single change will subsequently modify every artifact in the MTC by imposing intricate changes (adding, updating or deleting MTC elements) to its existing implementation.

Because of the strong dependencies that exist between the MTC artifacts, it is common that the *MTC developer* is confronted with several of these problems at the same time when he tries to directly evolve the artifacts of an existing MTC. We can say that these problems are highly related and if one of them affects the MTC, it is highly possible that the other problems affect the MTC as well.

1.1.1 Problem 1 (P1):The detriment of maintainability and understandability of MTC artifacts

When existing metamodels, models and transformations are directly adapted in order to support new concern-specific requirements, it is possible to reduce their understandability and maintainability. The detriment of maintainability and understandability in each type of MTC artifact is caused by the complexity of adapting the intricate relationships that exist between the elements of each artifact: 1) at the metamodel level, the *metamodels are polluted* by the inclusion of alien concepts into an existing metamodel, 2) at the model level, the *models become monolithic* due to the specification of every involved concern in a single model, and 3) at the transformation level, the complexity of transformations is increased by the addition of new *implicit intra-dependencies*.

Metamodel pollution

The concepts of the metamodel presented in Figure 1.2 are not sufficient to express authorization policies in the models. Therefore, one possibility is to directly extend the existing metamodel with security concepts. Figure 1.4 shows the metamodel extended with notions of authorization policies based on the Role Based Access Control (RBAC) model [SCFY96]. With these additional concepts it is now possible to specify an application that includes authorization policies. However, as shown in the figure, the simple metamodel presented previously becomes a more complex metamodel with an amalgam of concepts belonging to two different domains. Moreover, intricate relationships are created between the existing business concepts and the new security concepts, which creates a high coupling between the business and the security concern. Therefore, this situation clearly reduces the maintainability and understandability of the metamodel.

For instance, the extended metamodel presented in Figure 1.4 has security specific concepts such as *Role*, *Resource*, *Permission*, etc. These concepts must be merged with the concepts that belong to the original *Business metamodel*. In this case the business concepts *BusinessEntity*, *Attribute*, *Service* and *Association* have an inheritance relationship with the security concept *Resource*. These new concepts allow us to specify the idea of protection in the original concepts. Next, the *Action* concept must be extended in order to define the different actions that can be applied to a specific kind of resource (e.g. *Create* a *BusinessEntity* or *Read* an *Attribute*). Finally, an intricate set of relationships must be created between each type of *Resources* and its respective *Actions*.

Monolithic Models

The need for modularization and its benefits have been widely described and acknowledged in literature [Par72, Rea89, KLM+97]. Thus, whenever a specific

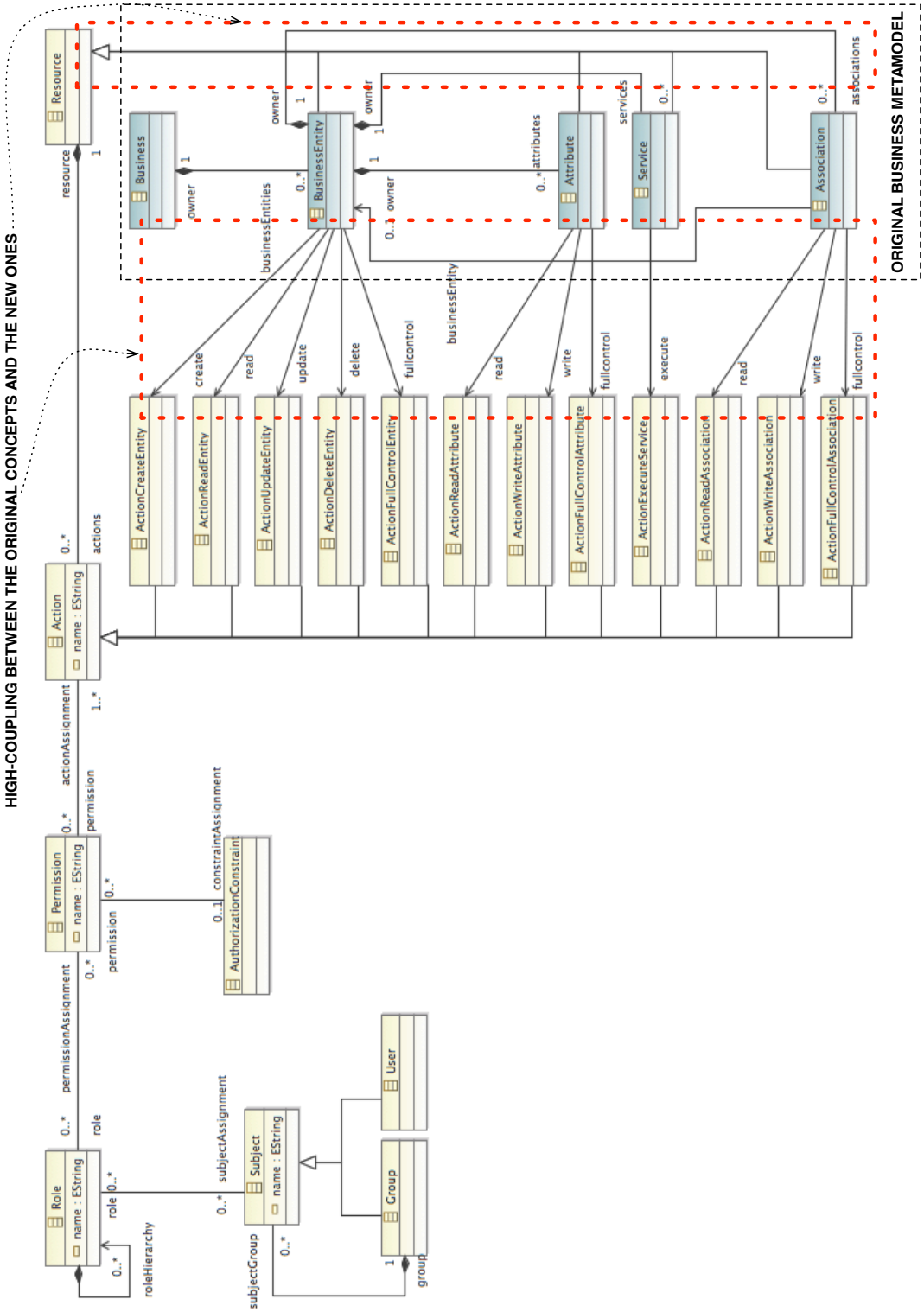


Figure 1.4: Extended Business metamodel

concern (e.g. authentication) requires to be specified, it is considered good practice to try to modularize it in order to avoid having every involved concern specified in a single *monolithic model*.

The extension of the metamodel with authentication concepts allows us to express authentication policies such as the *Permissions* and *Actions* for every *BusinessEntity*, *Attribute*, *Service* and *Association* that need to be protected. This situation increases the amount of information in the model and converts a simple model into a monolithic model. If more concerns are added to the metamodel and the whole application is specified in one single model, then this model will become incomprehensible and hard to maintain.

Figure 1.5 shows the updated monolithic model. This model contains the different business entities, their relationships, and all the security policies. In this model we have two policies: 1) the *Role* `User` is allowed to *Read* the *Attribute* `dueDate` of the *BusinessEntity* `Project`, and 2) the *Role* `Manager` that inherits the `User` policies is allowed to *Read* and *Write* the *Attribute* `dueDate` of the *BusinessEntity* `Project`. Even though this is an extremely simple model, these security policies increase the complexity in relation to the original business model. Therefore, a model with several entities, roles, and permissions will become difficult to maintain and understand.

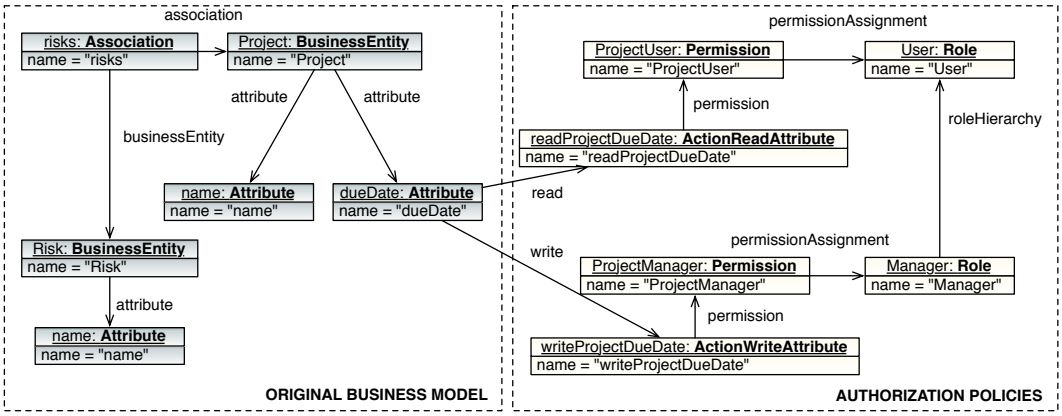


Figure 1.5: Extended Business model

Complex transformation implicit intra-dependencies

A model-to-model transformation is an intricate set of transformation rules that may have implicit dependencies among them. If implicit dependencies exist among the transformation rules and one of them is modified, then it will be necessary to carefully verify the other rules in order to validate that the implicit intra-dependencies are not broken.

For instance, Figure 1.6 presents an ATL transformation module [JK06]

with two rules[3]. The rule `BusinessEntity2Class` (lines 1-15) takes a *BusinessEntity* and produces a *Class*. The rule `Attribute2Field` (lines 16-32) takes an *Attribute* and produces a *FieldDeclaration*, its getter *MethodDeclaration* and its setter *MethodDeclaration*.

Between these two rules there are three implicit dependencies: 1) the binding of the feature *fields* in `BusinessEntity2Class` (line 8) and the output variable *field* generated by `Attribute2Field` (lines 20-23), 2) the binding of the feature *methods* in `BusinessEntity2Class` (lines 10-11) and the output variable *getter* (lines 24-27) generated by `Attribute2Field`, and 3) the binding of the feature *methods* in `BusinessEntity2Class` (lines 12-13) and the output variable *setter* (lines 28-31) generated by `Attribute2Field`. If a change is applied to any rule, then the developer must validate that the intra-dependencies are not broken.

```
1 rule BusinessEntity2Class {
2   from
3     entity   : BUSINESS!BusinessEntity
4   to
5     class    : JAVA!Class
6     (
7       name         <-  entity.name,
8       fields       <-  entity.attributes->union(entity.associations),
9       methods      <-  entity.methods->union(
10        entity.attributes->collect(e |
11          thisModule.resolveTemp(e,'getter')))->union(
12        entity.attributes->collect(e |
13          thisModule.resolveTemp(e,'setter')))
14    )
15  }
16 rule Attribute2Field {
17   from
18     attribute : BUSINESS!Attribute
19   to
20     field      : JAVA!FieldDeclaration
21     (
22       name     <-  attribute.name
23     ),
24     getter     : JAVA!MethodDeclaration
25     (
26       name     <-  'get' + attribute.name
27     ),
28     setter     : JAVA!MethodDeclaration
29     (
30       name     <-  'set' + attribute.name
31     )
32   }
```

Figure 1.6: Existing Transformation Rules

Figure 1.7 shows the extended ATL module. The rules in this module allow us to produce secured classes, getters and setters. In order to obtain this extended ATL module, several modifications are applied to the original

[3]A short ATL tutorial is presented in Appendix C.

module. First, the module needs to produce annotated[4] *Classes* with the list of all of the authorized roles (line: 14). The original rule was adapted in order to generate such annotations by iterating over all of the authorized *Actions* applied to the *Attributes, Associations* and *Methods* (lines: 16-29). Second, annotations are needed for getter (line: 40-44) and setter (line: 45-49) *MethodDeclaration* depending on the authorized *Actions*. This allows the execution of the getter *MethodDeclaration* only for *Read* actions and the execution of the setter *MethodDeclaration* only for *Write* actions. Finally, the actual *Annotation* for the *MethodDeclaration* getter is generated by the new rule `ActionReadAttribute2Annotation` (lines: 51-60). This shows the additional implicit dependency created between the rules `Attribute2Field` and `ActionReadAttribute2Annotation`.

In the original transformation there were three implicit dependencies between `BusinessEntity2Class` and `Attribute2Field` rules. In the extended transformation there are six dependencies, and this rule only manages one (i.e., `ActionReadAttribute`) of the twelve actions defined in the extended metamodel (Figure 1.4). In the presented rules in Figure 1.7 the new dependencies are: the dependency between the variable `annotation` (line 14) and the output variable annotation (lines 16-29), and the double dependency added between the `Attribute2Field` and the `ActionReadAttribute2Annotation` (i.e., one dependency for the setter and one for the getter). Additionally, there is a non-managed dependency between the role names specified in the `@DeclaredRoles` annotation in the generated *ClassDeclaration* and the role names specified in the `@RolesAllowed` annotation in the generated *MethodDeclaration* (i.e., an allowed role that can execute a method must be declared in the class). Therefore, the complexity of the original rule will be very high if the twelve *Actions* are implemented.

In summary, this example illustrates the impact of change on an existing model-to-model transformation by adding the security concern. The original transformation rules had to be adapted and new (implicit) dependencies were introduced. These dependencies increased the complexity of the transformations and made them harder to maintain and to understand.

[4]Java annotations are a JEE mechanism to offer authorization on its applications.

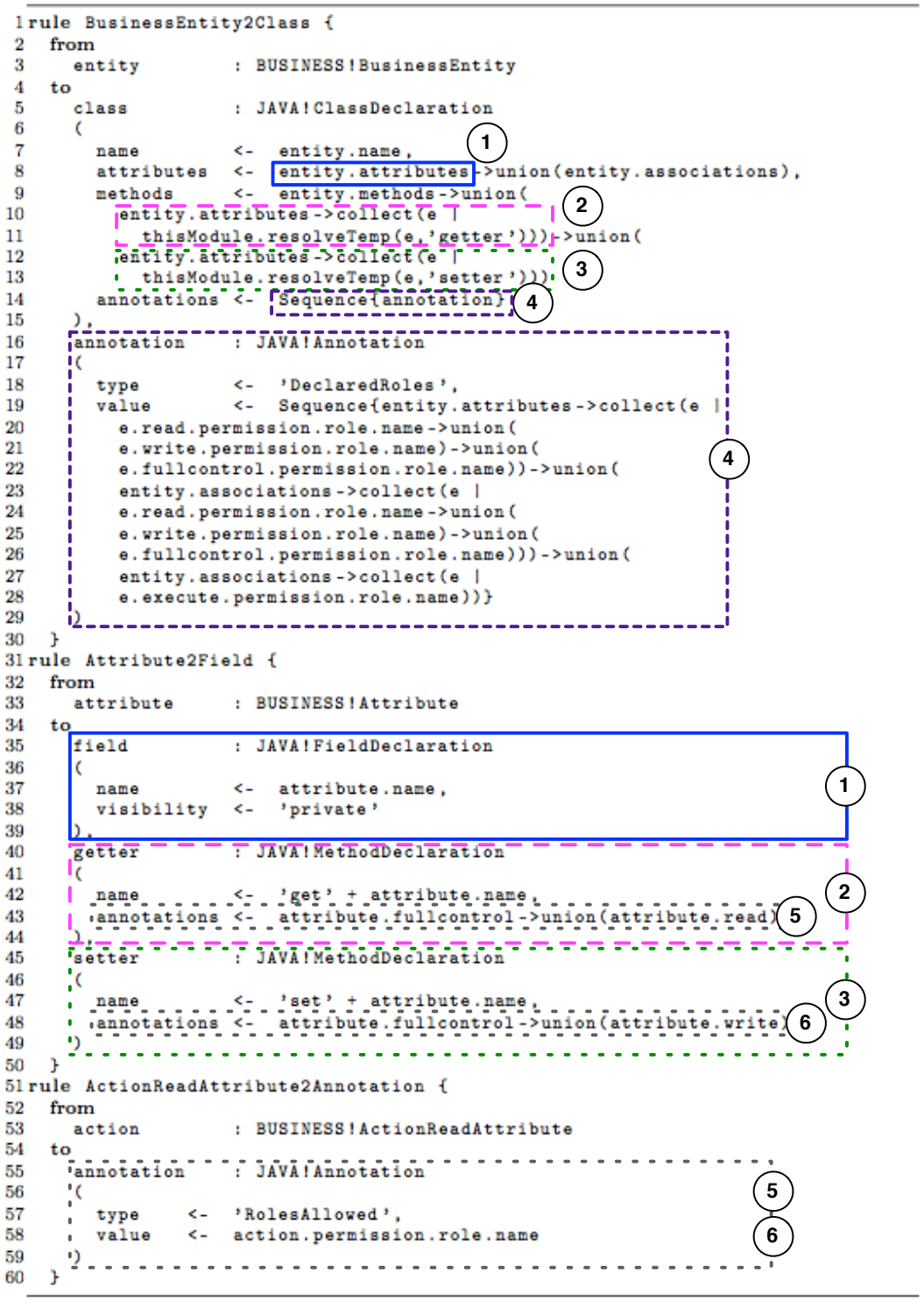


Figure 1.7: Extended Transformation Rule

- **Problem 1 (P1): The detriment of maintainability and understandability of MTC artifacts**
  The direct adaptation of an existing MTC in order to include a new concern will lead to the detriment of maintainability and understandability of its artifacts. In the case of adding alien concepts to a metamodel that clearly do not align with its existing concepts, it causes what we refer to as *metamodel pollution*. In the case of adding specifications of new concerns to an existing model, it causes what we refer to as *monolithic models*. Finally, in the case of modifying the existing transformation rules, it can create new *complex implicit intra-dependencies*. The fragility of the implicit dependencies inside a transformation rule makes the adaptation of a transformation a complex task.

1.1.2 Problem 2 (P2): Metamodel, Model and Transformations Co-evolution

There are strong inter-dependencies between the artifacts of an MTC. These dependencies exist between metamodels and models, and between metamodels and transformations. This means that if a metamodel is modified in order to include new concern concepts, it will be necessary to modify the models that conform to it, and the transformations that use it as source or target.

Metamodel and model co-evolution

In some situations where the metamodel is adapted, such as by modifying or eliminating existing concepts, the existing models will not conform to the new version of the metamodel. Therefore, to fix the problem, every model must be modified to correct the inconsistencies and restore the conformance relationship between the models and the metamodel. For example, if an attribute of a metaclass is eliminated, then every model must be modified by eliminating the attribute. This problem is known as *metamodel and model co-evolution* [Wac07, Fav03, CDREP].

For instance, Figure 1.8 shows a metamodel that has the metaclass *User*. This metaclass has two attributes: *firstName* and *lastName*. A model that conforms to version 1 of the metamodel has an instance of the metaclass *User* with *firstName* `John` and *lastName* `Doe`. If the metamodel is adapted to have only one attribute called *name*, then the model no longer conforms to version 2 of the metamodel. In order to fix this, the model must be modified by combining both attributes and assigning `John Doe` to the merged attribute.

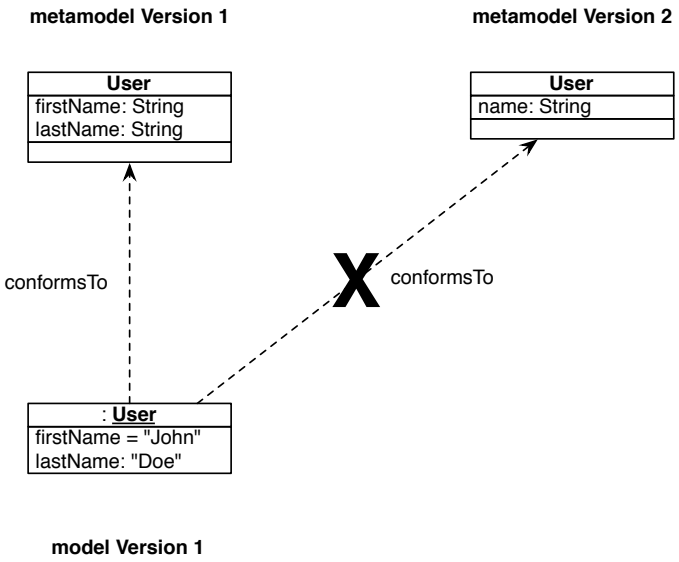


Figure 1.8: Metamodel and Model Co-evolution

Metamodel and transformation co-evolution

Furthermore, when a metamodel is adapted, the existing transformation rules need to be adapted as well. A transformation is highly dependent on the input and output metamodels, and if either of them are modified, then the transformation will need to be adapted as well. The adaptation of the transformations must be performed to support the new concepts or to fix the possible incompatibilities generated by the metamodel modification. The adaptation includes the addition of a transformation rule for every new concept in the metamodel, the modification of the existing rules when an existing concept is modified or when the new concepts have more complex relations with the original ones. We will call this problem *metamodel and transformation co-evolution*.

For instance, suppose we have a transformation that uses as input a model that conforms to the version 1 of the metamodel presented in Figure 1.8. If this metamodel is evolved, the transformation will not be compatible with the version 2 of the metamodel. In this case the transformation will expect a *User* element with two attributes (i.e., *firstName*, and *lastName*), and it will receive *User* elements with only an attribute called *name*. Therefore, when the source or target metamodels of a model-to-model transformation are modified, the transformation will need to be modified as well.

- **Problem 2 (P2): Metamodel, Model and Transformations Co-evolution**
  Adapting a metamodel by removing or modifying its concepts could break the conformance relationship with its existing models as well as the transformations that use it as source or target. When this happens, the models must be evolved to repair the conformance relationship and transformations must be adapted.

1.1.3 Problem 3 (P3): Ripple effect

The *ripple effect* is the main problem when an MTC is adapted to include a new concern. This problem subsumes the problems mentioned before and replicates them in a chain reaction through the whole MTC. As explained before, there are strong dependencies between the MTC artifacts. These dependencies are between metamodels and models, metamodels and transformations, and between each transformation step and those which follow it. In other words, when an MTC metamodel or transformation is modified to include a new concern, the subsequent metamodels and transformations must be adapted as well. Moreover, the dependencies between the different rules and steps are not explicitly defined. Therefore, if a change impacts a metamodel or a transformation, then the complexity and cost of maintaining the consistency among all the artifacts will increase. Therefore, every step should be carefully adapted to assure that the chain is not broken and does indeed support the new concepts [KKS07].

For instance, suppose that we have a four-step MTC that uses a high-level model that conforms to a Business metamodel and produces Java applications. These MTC steps are:

1. The first step is to transform the high-level business model into an architectural model that conforms to an Architectural metamodel that contains concepts such as *Layers* and *Communications* that relate the *Layers*.

2. In the second step, the architectural model is transformed into a JEE model that conforms to a JEE metamodel that contains concepts such as *EntityBeans* and *SessionBeans*.

3. In the third step, the JEE model is transformed into a Java model that conforms to a Java metamodel that contains concepts such as *Classes*, *Methods* and *Annotations*.

4. In the final step, the Java model is transformed into Java code.

Now, suppose that security concepts are added to the Business metamodel in order to support authorization policies. This change will yield all the problems mentioned before. This means that the Business metamodel will be polluted with security concepts, the models will not conform to the metamodel, the transformation will not support the new models, the models will be monolithic, resulting in a high probability of breaking the transformations intra-dependencies. Hence, in order to translate the security concepts into the next level (i.e., architectural level), we will need to include security concepts in the architectural metamodel as well. Although we identify how to represent authorization concepts at the architectural level and have adapted the Architectural metamodel, we will end again having the same problems as in the business level (e.g., metamodel pollution). This situation will repeat one after the other in each transformation step. Therefore, if one of the transformation steps is modified to support new concepts, then every following step in the transformation chain must be modified as well. A simple change in the first set of artifacts could require changes in all of the artifacts in the MTC in a *ripple effect*. Figure 1.9 shows how a change in the highest level metamodel triggers changes throughout the whole transformation (dashed arrows represent the *ripple effect*). The only level that is not affected by the change is the lowest level because it has all the necessary concepts to enforce security (i.e., Java annotations). More details about this problem are presented in Section 3.4.1.

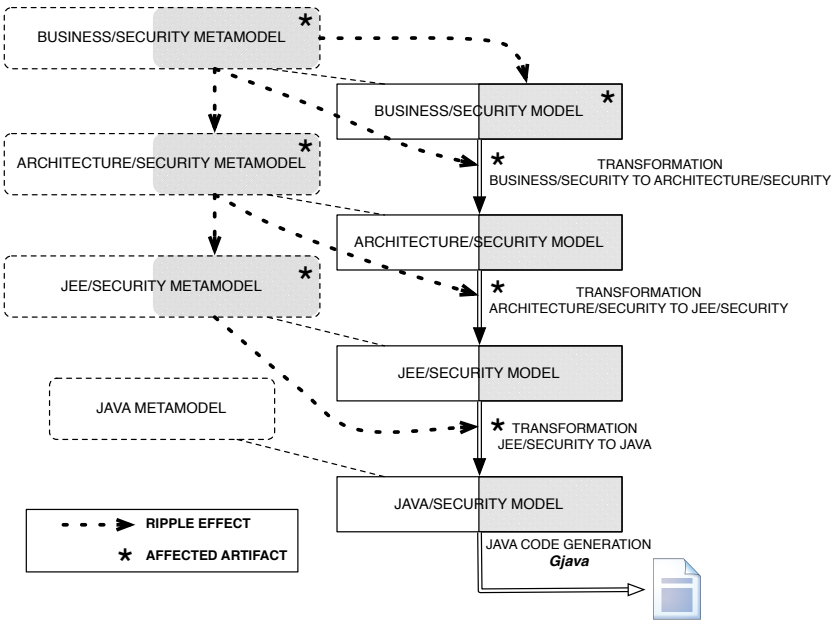


Figure 1.9: Ripple effect

- **Problem 3 (P3): Ripple effect**
  The most complex problem that is originated by the addition of a new concern in an existing MTC is the *ripple effect*. This means, that if an artifact is modified by the addition of a new concern, every subsequent artifact must be modified as well.

1.2 Research Goals

Having in mind the problems presented in the previous section, we envision an approach that avoids the modification of the existing assets (i.e., metamodels, models and transformations) when an existing MTC is evolved.

1.2.1 General Goal: Non-invasive evolution of an MTC

The general objective of our research is to propose an approach that avoids changing the original MTC. This means that the original metamodels, models and transformations are not changed when the new concern is introduced. By avoiding the direct modification of the original MTC artifacts, we avoid all the problems mentioned before (See Problems P1 - P3). This prevents the maintainability and understandability detriment of the original MTC assets. Additionally, the conformance relationship between models and metamodels is not broken because the metamodel remains unchanged. Similarly, the transformations remain compatible with the metamodel. Moreover, because no artifact is modified, the *ripple effect* problem does not arise. In other words, we can say that the original MTC remains oblivious of the change.

1.2.2 Specific Goals

In order to perform a non-invasive evolution of an existing MTC, we want to maintain the changes modularized without performing any modification to the original MTC while following the *Separation of Concerns* principle (SoC) [Dij82]. The SoC principle states that the different aspects of a system should be specified separately. In other words, we envision an approach where the changes are modularized and are specified using concern-specific concepts. Moreover, the proposed approach must offer a mechanism to specify the concern-specific requirements at the correct level of abstraction. Finally, the proposed approach must provide a reusable mechanism to identify the elements affected by the change. This mechanism must be independent of the MTC, allowing the original artifacts to remain oblivious of the changes. In addition, the mechanism must be able to introduce these changes into the MTC

at the most suitable moment. The most suitable moment is selected by identifying the place in the MTC where the original artifacts remain unchanged. From these ideas we extract a set of specific goals:

Goal 1 (G1): Concern-specific modularization

The first specific goal is to offer an approach that allows the *MTC developer* to encapsulate the introduced changes in concern-specific modules. These modules must contain the concern-specific changes that support the new requirements and affect the existing metamodels, models and transformations. If our approach offers such modularization, we will keep the original assets unmodified and we will improve the maintainability and understandability of the MTC by following the SoC principle. Additionally, as we maintain the original artifacts unchanged, we do not need to worry about metamodel, model and transformation co-evolution problems or triggering a *ripple effect* through the MTC steps. This goal will help us to avoid Problems P1, P2 and P3.

Goal 2 (G2): Specifying the different concerns at a high-level of abstraction

The second specific goal is to allow the *application modeler* to specify multiple application concerns at a high-level of abstraction. In other words, each concern must be specified at the highest possible level of abstraction while using concern-specific concepts. This will allow the *application modeler* to create multiple concern-specific specifications where the multiple application requirements are modeled. Using high-level metamodels gives the domain and concern experts the appropriate abstractions to define the application specification, improving the maintainability and understandability of the MTC. This goal will avoid Problem P1.

Goal 3 (G3): Enabling an oblivious mechanism to integrate new concern-specific requirements

The third specific goal is to provide an oblivious mechanism that allows *MTC developers* to integrate the new concern-specific requirements. This mechanism must be metamodel independent, enabling the *MTC developers* to reuse it for adding non-supported concern-specific requirements to existing MTCs. This mechanism should allow *MTC developers* to relate the new requirements with the existing elements that they impact. The existing elements must be oblivious of the new requirements. Finally, this mechanism must transparently integrate the new requirements without modifying the existing artifacts. This will avoid triggering a *ripple effect* through the MTC steps (See Problem P3).

1.3 Approach

The problems we discussed originate from the impact of change on the original assets (i.e., metamodels, models and transformations). We want to apply the SoC principle to evolve an MTC. Additionally, we want to apply the SoC principle in the context of MTC evolution in order to gain the advantages obtained in other contexts such as Aspect Oriented Programming (AOP) [KLM+97].

Figure 1.10 presents a general schema of our approach. In our approach we modularize the changes in a new concern-specific MTC. This new MTC should be added next to the existing one, using a high-level concern-specific model as input and producing a low-level concern model as output. This means that the required application can be specified using both the original model and a new concern model. Both high-level models (i.e., existing model and concern model) should be aligned in order to identify corresponding elements in each high-level model. This means that the new high-level model must be aligned with the original one by using a *Correspondence Model* (CM) [BBDF+06]. A CM explicitly describes the correspondence relationships among the elements of different models.

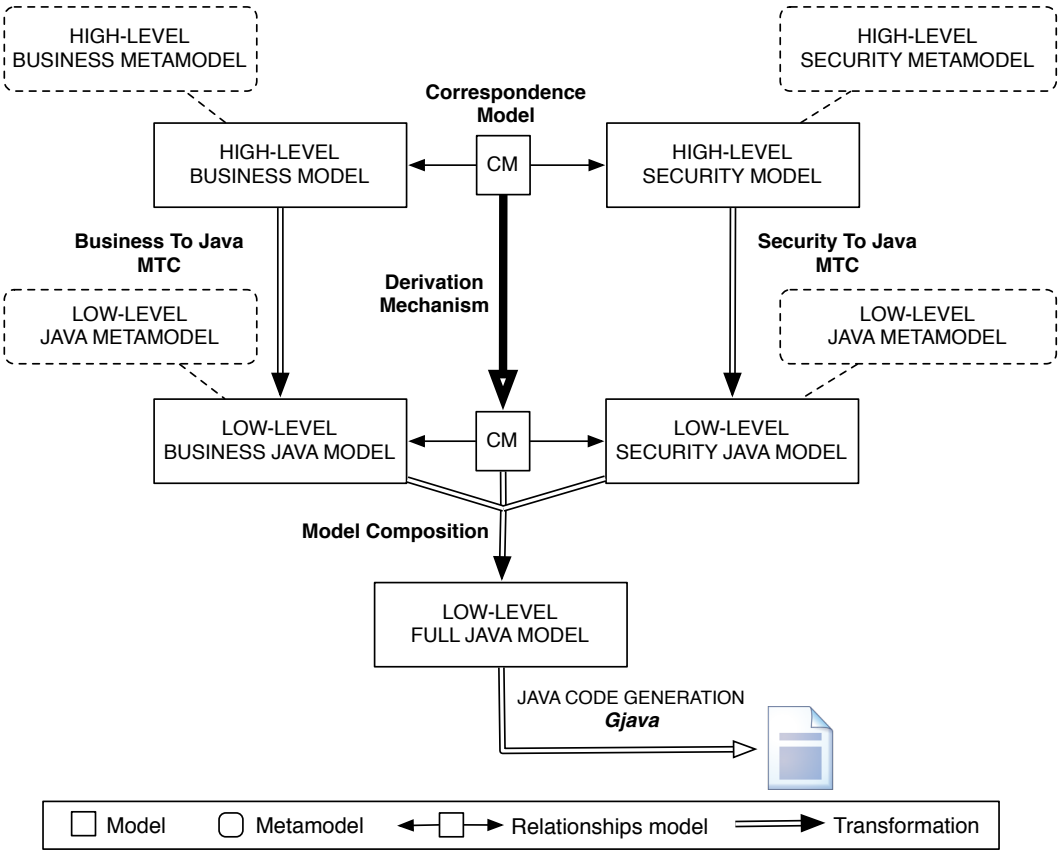


Figure 1.10: General Schema

The two MTCs produce two low-level models, that are the translation of the high-level specification into a common low-level metamodel (e.g., Java). Both models are complementary low-level models that can be composed using a common composition mechanism because both models conform to the same metamodel (e.g., Java metamodel). This type of composition is called a homogeneous composition and is much simpler to accomplish than a heterogeneous composition which involves different metamodels. A homogeneous composition only requires identifying corresponding elements and does not require a semantic alignment. Additionally, a homogeneous composition mechanism can be reused by any concern that can be expressed using the same metamodel. When we compose both models, we obtain a full low-level model with all the requirements in it. However, in order to compose both low-level models, the corresponding elements that exist between both models must be identified.

The main challenge is to define a mechanism that automatically derives the new correspondence relationships between the low-level models, keeping in mind that every step of the MTC increases the complexity of the models by adding elements at each step. Commonly, these new elements are implementation details of the application.

We propose a *correspondence derivation mechanism* that uses the high-level correspondence model to automatically obtain a correspondence model between the low-level models.

As said before, when we reach the lowest level, the models conform to the same metamodel. The derived correspondences between the low-level models are used to identify the corresponding elements and to perform the composition.

In order to use the presented approach, one must:

- *facilitate the use of different metamodels* with suitable constructs in order to independently express the different involved concerns following the SoC principle.

- *provide a high-level alignment between different concern models that conform to different metamodels.* The alignment should allow the reasoning about the relationships between the different models. The main objective of aligning the high-level models is to reduce the amount of manually defined alignment relationships. This is because at a high-level of abstraction fewer concepts are defined than at a low-level of abstraction.

- *maintain an alignment between the added concern-specific MTC and the original MTC* in order to derive the correspondence relationships between the low-level models to be able to use them to perform a composition.

- *use a common composition mechanism* to integrate the concern model with the existing model when both models conform to the same common

low-level metamodel (e.g. Java metamodel). Moreover, if an additional concern can be specified using the common low-level metamodel, then the composition mechanism can be reused.

1.4 Approach Scope

As was introduced before, the main goal of our research is to provide a non-invasive approach to evolve an existing MTC by adding a unexpected concern. In other words, we envision a solution to the problem of adding complex concerns to an existing MTC and avoiding the modification of the existing assets. Additionally, we want to be able to encapsulate these complex concerns in modules following the SoC principle (G1).

The type of changes that we want to introduce to the existing MTC must be related with a concern that was not expected in the development of the existing assets (i.e., metamodels, models and transformations). As we want to specify the added concern at a high-level of abstraction (G2), we will need to define a self-contained concern metamodel. This means that all the concepts required to specify the concern for the required application must be in the metamodel. There are many possible concerns that can be modularized such as: security, monitoring, user interface, business rules, etc.

Our approach, does not include support for evolution caused by small changes in the existing metamodels or transformations. The changes to the high-level metamodel usually are caused by the evolution of the problem domain. When this happens the high-level metamodel must be adapted and in the best-case scenario extended with new concepts. Our approach focuses only in adding concern concepts that do not fit with the concepts of the existing high-level metamodel.

When the low-level metamodel or the transformations require to be modified, it is usually caused by the evolution of the solution domain. For instance when the technological platform evolves or new requirements related to it are added. When the changes that are required to perform are fully related with the existing infrastructure, our proposal in not suitable for evolve the MTC artifacts. We want to provide support for independently modeling a new concern and integrating it automatically to the existing infrastructure (G3).

In summary, if the required changes are simple, such as adding a new attribute, or a couple of metaclasses, our approach is not suitable for it. Additionally, if the changes only require to modify the transformations or the low-level metamodel, our approach is not suitable either. For this type of changes, there are some works that we will present in the next chapter.

1.5 Contributions

The main contributions of the research presented in this dissertation,can be summarized as follows:

- **A novel strategy to perform a non-invasive evolution of model transformation chains**
  The main contribution of our approach is a strategy to perform a non-invasive evolution of an existing model transformation chain. This evolution is limited to the introduction of new concerns. In other words, we do not consider MTC re-factoring or the introduction of new functionality within the same concern. The approach modularizes the changes by adding an additional concern-specific model transformation chain. It allows the specification of a high-level concern model with concepts that do not belong to the original model transformation chain. This new concern model is consequently transformed into a low-level model and composed with the original low-level model. This strategy avoids changes to the original MTC, and encapsulates the new concern concepts and the new required transformations in a concern-specific transformation chain. This was published in [YCDW09a, YCDW09b, YCDW10a, YCDW10b].

- **A mechanism to automatically derive low-level correspondence relationships**
  We provide a mechanism that automatically derives correspondence relationships between two target low-level models. These relationships are generated by analyzing the traceability models generated by the transformations and a correspondence derivation model that defines how and which elements should be related. Even though the mechanism was originally developed to work with the same target metamodel, we have also extended it to work with different target metamodels. This was published in [YCDW09a, YCDW09b, YCDW10a, YCDW10b]

The secondary contributions of the research in this dissertation are the following:

- **An analysis of the strategies that can be used to evolve an MTC**
  We studied multiple works seeking out strategies that allow the specification of several concerns and automatically compose them to produce a full application. We translated the ideas of these works to the field of MTC evolution. We compared the chosen strategies with key criteria in order to select the most suitable strategy. The analysis of the possible strategies was presented in [YCDS08].

- **Tool support**
  We have developed a set of Eclipse plug-ins and extensions that help to use our approach in the evolution of MTCs. These tools are available at `http://qualdev.uniandes.edu.co`. The tools are:

  1. *Correspondence Modeler:* a modeler to specify correspondence relationship between two high-level models that conform to two different metamodels.
  2. *Correspondence Derivation Modeler:* a modeler to specify restrictions on deriving correspondence relationships. Additionally, this plug-in generates the transformation required to derivate the correspondence relationships.
  3. *Composition Generator:* a plug-in that generates a composition transformation.
  4. *Correspondence Checker Generator:* a plug-in that generates a checking transformation.
  5. *Traceability Processor:* a plug-in that analyzes a set of tracing models and generates its transitive closure.
  6. *ATL traceability mechanism:* we extended the ATL-VM to offer advanced traceability mechanisms. This extension was presented in [YW09].

1.6 Outline of the Dissertation

This section presents an overview of the structure of this dissertation.

Chapter 2: MDE, Evolution in MDE, and Separation of Concerns in MDE

This chapter gives an in-depth explanation of MDE. The main concepts of MDE, such as model, meta-model and model transformation, are discussed in detail. Additionally, we present the use of DSMLs in multi-modeling environments and composition techniques. Finally we discuss the evolution of MDE.

Chapter 3: Evolving a Model Transformation Chain

In this chapter, we present an analysis of several strategies that can be used to evolve a model transformation chain. These strategies are analyzed using a set of key criteria that allow the identification of the most suitable strategy.

Chapter 4: Correspondence Relationships Derivation

This chapter presents the strategy we use to derive a correspondence model between the target low-level models. This mechanism uses traceability models and a correspondence derivation model that restricts the generation of low-level correspondence relationships.

Chapter 5: Correspondence Relationships Resolution

The objective of this chapter is to present the correspondence relationships in detail, their semantics and how we use them to relate models at different levels of abstraction. Additionally, we present our approach to resolve the relationships.

Chapter 6: Tool Support

This chapter provides in-depth discussion of the provided tools and how they can be used to support the evolution of existing model transformation chains. First, the global architecture of the tools is explained. Then all the components that are necessary to correctly extend a MTC. Following that, the required tasks in order to extend a MTC are discussed. Finally, the limitations and future directions of the tools are discussed.

Chapter 7: Validation: Evolving Transformation Chains

This chapter presents the validation of our work. This validation is presented with the help of two case studies. The first case study adds an authorization mechanism to the existing MTC. The second case study adds a web layer to the same existing MTC.

Chapter 8: Conclusion

This chapter concludes this dissertation with an evaluation of the approach we have presented. An overview of our contributions is given and future research directions are explored.

Chapter 2

MDE, Evolution in MDE, and Separation of Concerns in MDE

2.1 Introduction

We dedicate this chapter to presenting the notions and concepts that are behind the evolution of a Model Transformation Chain (MTC), with the objective of fully understanding the context where our work is delineated, the problems that we face, and the main characteristics of our approach. Simultaneously, we present some current research related with the presented subjects.

We divide this chapter in three main parts where we group the necessary concepts to comprehend our work. In the first part, we present the concepts to understand the context of our research. In the second part, we introduce current research that focuses on solving the main problems in the area of MDE evolution. Finally, in the third part we describe the concepts behind the Separation of Concerns (SoC) principle applied to MDE that will allow for comprehending our proposed solution.

Figure 2.1 presents a schema of the distribution of this chapter, relating its sections with the elements of our approach. In Section 2.2, we introduce the ideas behind Model Driven Engineering (MDE) and how it increases the abstraction level by using models and metamodels. Additionally, in this section we present model transformations and how models, metamodels and transformations are combined in an MTC in order to generate applications. Finally, in this section we present the ideas of model traceability. The objective of this section is to present the background knowledge required to appreciate our work. In Section 2.3 we present the main problems in the area of MDE evolution and current research used to tackle these problems. These works are directly related with our work in MTC evolution. Finally, in Section 2.4 we show how the SoC principle is used in MDE in order to specify multiple concerns in different models and then how to compose them in order to obtain the final application. This section presents background material that inspired

us to formulate our solution.

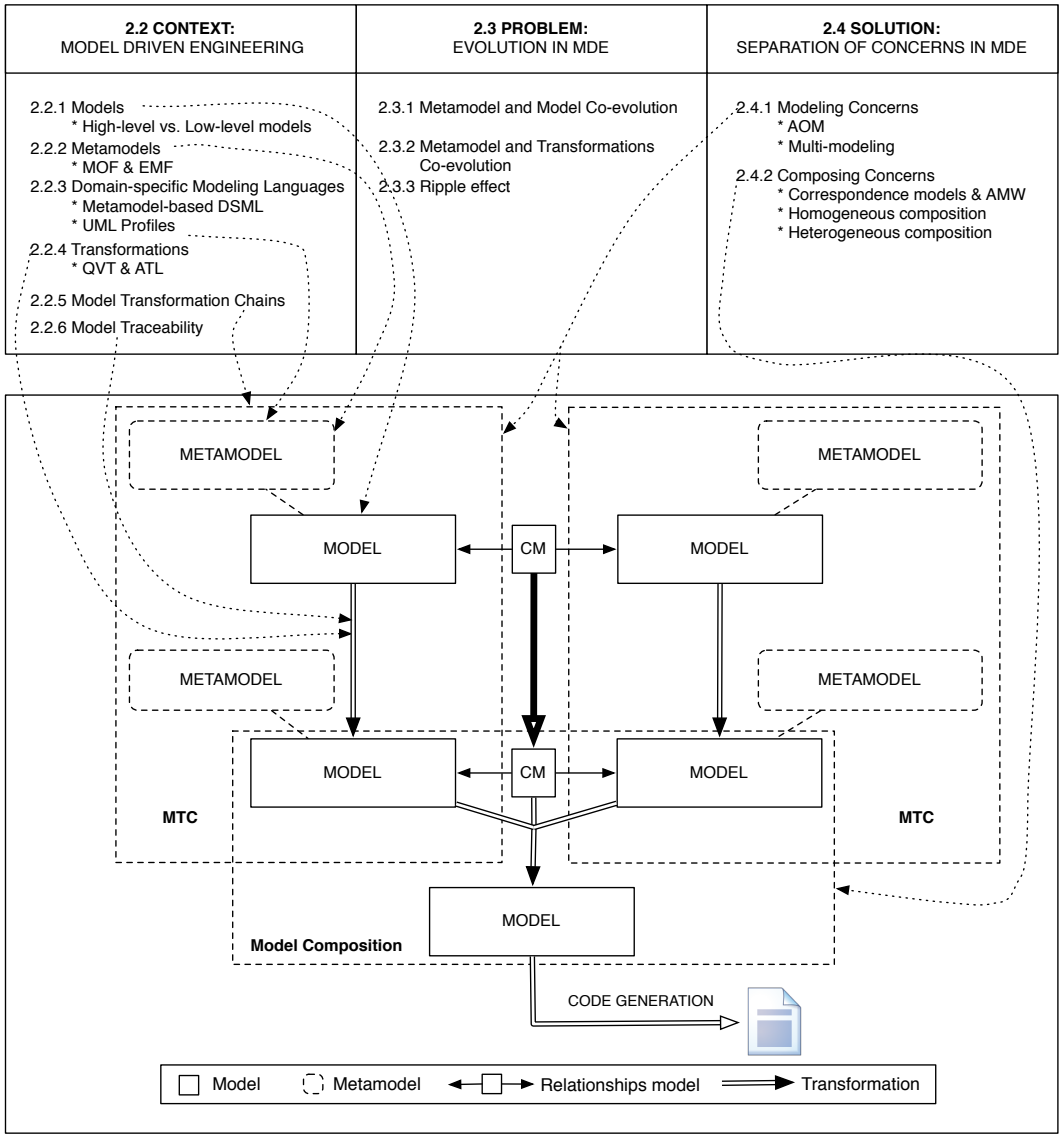


Figure 2.1: Structure Chapter 2

2.2 Context: Model Driven Engineering

Model-Driven Engineering (MDE)[Sch06] has reduced the gap between problem and solution domains by using concepts that belong to the problem domain to specify a system, and by using automatic transformations to bring this specification to the solution domain.

MDE uses *models* to increase the level of abstraction. MDE considers

models as first class entities in the development process [BJRV05]. This means that they are main artifacts throughout the entire development process. This is a fundamental difference in comparison with other movements (e.g. code-centric approaches) that use only models as mechanisms of representation, documentation, and communication. In the next section we will present several definitions of a *model* in order to identify a common notion of it.

2.2.1 Models

There are several definitions of what a *model* is. Usually, models are used to analyze the specified system or to generate the actual implementation of the required system. It is almost impossible to have a universally accepted definition[Lud03].

The OMG states that *"A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text."* [MM03]. This definition presents the general idea of what a model is and how it is represented.

Bézivin states that *"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system."* [BG01]. In this definition it is important to notice that a model is a representation of the system and allows to reason about the actual system.

There are two types of models based on their analytical or synthetical intention [MFB09]. On the one hand, an *analytical* model allows us to make predictions or inferences about the system. On the other hand, the intention of a *synthetical* model is to generate the system from it.

From these definitions we can gather that a model represents a system and can be used to analyze or to make inferences over the system. Additionally, the model can be used as a mechanism to specify and generate the required system.

In our research, we use models as mechanisms to generate a system. Consequently, it is important for us to differentiate the kind of models based on the amount of technological details in them. For this reason we divide them in *high-level models* and *low-level models*.

High-level vs. Low-level models

In MDE several types of models are used. A specific division used in the generation of applications using models is related to the amount of computational and technological platform details existing in the models. This division is used in the MDE initiative of the OMG that is called *Model Driven Architecture* (MDA) [MM03]. The main goals of MDA are portability, interoperability and reusability through architectural separation of concerns.

MDA promotes the use of three types of models related to the level of abstraction and the amount of technological details in it:

1. *Computation Independent Model* (CIM): this is the model at the highest level of abstraction focused on the environment and the requirements of the system; the details of the structure and processing of the system are hidden or undetermined. In this model the system requirements are expressed using the domain concepts as vocabulary. The goal of the CIM is to reduce the gap between the business experts and the system design experts.

2. *Platform Independent Model* (PIM): the focus of this model is on the operation of a system while hiding the details necessary for a particular platform. A platform independent model does not change from one platform to another. This model describes the actual system without the technological platform details.

3. *Platform Specific Model* (PSM): this is the model at the lowest level of abstraction. In the PSM the system is specified by combining all of the elements specified in the PIM with the addition of technological platform specific details.

The important notion behind this division is that each kind of model is at a different level of abstraction, starting from the highest level (CIM) to the lowest level (PSM). The level of abstraction is determined by the amount of computational and technological platform details in the models. Although this division is part of the MDA specification, in real life it is difficult to differentiate between these three levels. Because of this difficulty, in this dissertation we divide the models into two types: 1) the first group that we call *high-level models* which are specified using concepts that belong to the problem domain, and 2) the second group that we call *low-level models* which are specified using concepts that belong to the solution domain. In actual MDE implementations, multiple levels are used starting from a high-level model that is specified in terms of problem domain concepts (e.g. Entities, Services), and transformed in several steps into a low-level model that is specified in terms of the solution domain concepts (e.g. Classes, Methods). The lowest-level model has all the requirements specified in the high-level models plus the technological platform details in it, and are expressed in terms of the solution domain concepts.

In order to be able to create a model, it is necessary to use a modeling language. This modeling language is expressed in the form of a *metamodel*. A metamodel is a model and is also a model of a modeling language [CSW05].

2.2.2 Metamodels

Models are specified in a modeling language, and in MDE a modeling language is defined by a *metamodel*. Favre states *"A metamodel is a model of a language of models"* [Fav04]. A metamodel describes the abstract syntax of the modeling language, using the concepts of the domain and their relationships.

A metamodel should be part of a *metamodeling architecture*, which enables a model to be viewed as a model, and in a similar way it is itself described by another metamodel. This allows all metamodels to be described by a single metamodel named *meta-metamodel* [CSW05]. The OMG proposes a standard metamodel architecture that consist of four levels. These levels are depicted in Figure 2.2.

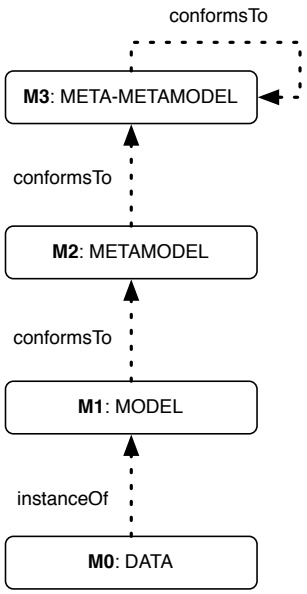


Figure 2.2: The 4-level OMG metamodeling architecture

Moreover, the OMG proposes a standard meta-metamodel named *Meta Object Facility* (MOF) [Obj01] that allows the definition of metamodels. A framework that follows this standard is the *Eclipse Modeling Framework* (EMF) [SBPM09]. We use EMF extensively in our research and in the implementation of the tools. In the next subsection, MOF and EMF are briefly explained.

MOF & EMF

The OMG proposes the *Meta-Object Facility* (MOF) standard as a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata driven systems. MOF mainly serves as the platform-independent metadata management foundation for MDA [Obj01].

Eclipse Modeling Framework (EMF) [SBPM09] is a modeling framework and code generation facility for building tools and other applications based on a structured data model. Although EMF intends to comply with MOF, its metamodeling language *Ecore* differs from the MOF specification. Figure 2.3 shows a simplified subset of the Ecore meta-model.

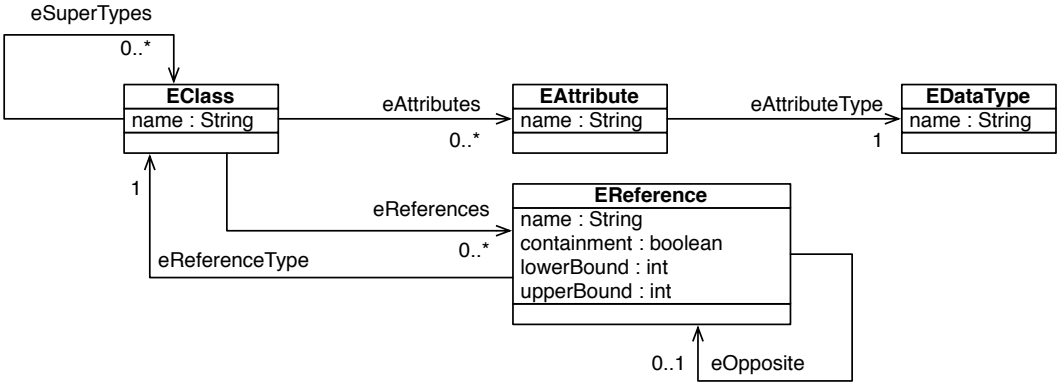


Figure 2.3: Ecore kernel metamodel (source: [SBPM09])

The key EMF concept is *EClass*, which is a concept that represents a metaclass and is the main building block to define an Ecore metamodel. Additionally, EMF has the concepts *EAttributes* and *EReferences*. The difference between *EAttribute* and *EReference* is that the type of an *EAttribute* is always a primitive type, such as string, boolean or integer, while the type of an *EReference* is always an *EClass*. Moreover, as an implementation framework, EMF allows the behavior of *EOperations* to be specified in Java.

The models used in MDE are usually defined by a *Domain Specific Modeling Language* (DSML) [TK05]. A DSML increases the level of abstraction and gives suitable concepts to domain experts for specifying an application close to the problem domain (e.g., insurance domain, finance domain). The domain concepts and their relationships that are available in the DSML are specified in a metamodel.

2.2.3 Domain Specific Modeling Languages

A *Domain Specific Language* (DSL) [VDKV00] is a programming language specialized in a particular problem domain or technological platform. In contrast, General Purpose Languages (GPLs) are created to solve problems in many domains. For instance, SQL is a DSL for the relational databases domain, in contrast to Java which is a GPL.

In MDE, there are approaches that follow the DSL idea by using models instead of textual languages. These approaches use Domain-Specific Modeling Languages (*DSMLs*) [TK05], which increase the level of abstraction and give

domain experts concepts that belong to the problem domain (e.g., insurance domain, finance domain) in order to specify an application. Additionally, because domain experts are familiar with the concepts of the domain, they can quickly learn, use, and evolve the language. Finally, The use of DSMLs reduces accidental complexities that are introduced if an application is specified using an unsuitable language. For instance, trying to specify technological platform specific properties at domain level.

There are two main methods of defining a DSML. The first method starts from scratch defining a new metamodel. This is a common method that uses a minimalistic approach to develop a language. The second method takes an existing language and extends it by including domain concepts. Usually, this method takes a General Purpose Modeling Language (GPML) such as UML and adds domain concepts to it.

Metamodel based DSML

When DSML is defined from scratch, it is specified using a *metamodel* that describes all the domain concepts and their relationships. The language developers specify the abstract syntax of the modeling language using the metamodel. Additionally, they need to define the concrete syntax and the semantics of the DSML. There are several DSML definition environments that help the language developers to create languages and tools. Among these are some commercial tools such as MetaEdit+[Met10], open source such as GEMS[GEM10] and openArchitectureWare[ope10a], or academic such as GME[GME10].

The main critique of this method is the necessity to create a language from scratch, including its editors, compilers, and tools. Additionally, the *application modelers* will need to go through the extra effort of learning a new language. Instead, starting from scratch allows for the specification of a language without inheriting any problems of an existing language. The new DSML is created only with necessary concepts following a minimalist vision of the language.

UML Profiles

Another approach to create a DSML is to use a GPML such as UML and extend it with domain concepts. UML offers a mechanism named *UML Profile* that is commonly used in the DSML field. A Profile provides a lightweight extension mechanism to customize the semantics of UML for particular domains or platforms. This means that each Profile follows the UML semantics, but introduces domain specific semantics that narrow down existing UML semantics. A Profile is defined by using stereotypes, tag values, and constraints that are applied to specific model elements, such as Classes, Attributes, Operations, etc.

There are several domain specific extensions defined using UML Profiles such as: SysML[Obj10] used for modeling system engineering applications, or MARTE[Obj09b] used for modeling real-time and embedded applications.

The main critique of using UML Profiles comes from transfering the existing problems of UML to the new DSML. UML offers a large amount of commercial and non-commercial tools, and it is a well-known language that has a considerable amount of practitioners.

Either of the two options (i.e., metamodel based or profiles) can be used to specify models in order to represent or to generate a system. Whichever model will conform to a DSML metamodel or to the UML metamodel plus the profile. In this dissertation we choose to work with the minimalist approach (from the language point of view) of defining our DSML from scratch. We prefer to have absolute control over all the concepts in our metamodels and do not need to understand and use the more than 200 classes of UML.

Now that we have the notions of models, their relationship with metamodels and how they are defined using DSMLs, we will explain how models are processed by *Model Transformations* in order to produce new models or to generate source artifacts from them.

2.2.4 Model Transformations

Model Transformations are a key component in MDE that process models in order to generate several types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations [Sch06]. Transformations use input models to generate source artifacts or new models that describe a different representation of the modeled system. They are used to navigate models using the structure defined by their metamodels. Thus, transformations are defined in terms of metamodel concepts and they can process any model that conforms to the metamodels used by the transformation.

Transformations can be divided in two major categories: *model-to-text* and *model-to-model* transformations. The model-to-text transformations category takes a model as input and produces text artifacts using visitor-based or template-based approaches. We can find languages in this category such as Acceleo [Obe10], and Xpand[ope10b]. The model-to-model transformations category produces a new model that conforms to a target metamodel. We can find languages in this category such as ATL [JK06], and QVT[Obj09a].

Figure 2.4 presents a schema of a simple model-to-model transformation, where a transformation is defined in terms of the source and target metamodels. This means that any model that conforms to the *source metamodel* can be processed by the *transformation engine* using the *transformation definition*, and that it will produce a new model that conforms to the *target metamodel* [JK06, Obj09a].

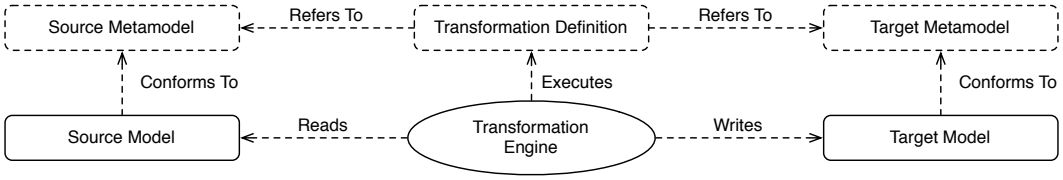


Figure 2.4: Model-to-model transformation basic schema (source: [SBPM09])

The model-to-model transformation standard proposed by the OMG is the *MOF Query/View/Transformation* (QVT) transformation language. Although there are some implementations of QVT languages (e.g., medini QVT, QVTd and SmartQVT), they are not broadly used currently mainly because of their recent stable implementations and lack of documentation. As an alternative, some "QVT-like" transformation languages are becoming more commonly used in research and industry projects. One of these QVT-like languages is the *Atlas Transformation Language* (ATL) [JK06] that is one the most widely used transformation languages in the MDE community. ATL is the transformation language that we use in our implementation and examples. These two languages are presented in the next subsection.

Transformation Languages: QVT and ATL

On the one hand, *MOF Query/View/Transformation* (QVT) is the standard transformation language proposed by the OMG to process MOF based models [Obj09a]. As previously stated, model transformations are a critical component of MDE. For this reason, in 2002 the OMG asked for a proposal in order to seek a standard compatible with the MDA standard. Consequently, the first complete version of the QVT standard was published in 2008. QVT is divided in three main languages: *QVT Operational*, *QVT Relations*, and *QVT Core*. *QVT Operational* is a language that allows the definition of a transformation using an imperative approach (e.g., SmartQVT). QVT Relations is a declarative language in which bidirectional mappings are defined between two metamodels (e.g., medini QVT, QVTd). Finally, QVT Core is a low-level declarative language.

On the other hand, it is possible to say that ATL is the *de facto* standard due to its user base. ATL is a hybrid transformation language that supports declarative and imperative constructs. The main declarative construct of ATL is a *matched rule*. This kind of rule consists of a left-hand side that uses a source pattern to identify matched elements and a right-hand side that produces a target pattern for every match. Additionally, ATL has two imperative constructs: *called rule* and *action block*. A *called rule* must be explicitly called similarly to a procedure. These kinds of rules have a target pattern that creates an element every time that the rule is called. Finally, *Action blocks* are

sequences of imperative instructions.

A transformation rule in ATL is specified in modules. An ATL module can be expressed as a model that conforms to the ATL metamodel. The ATL transformation schema is presented in Figure 2.5. Although this schema is similar to the generic model transformation schema that was presented in Figure 2.4, it is important to notice that an ATL transformation module is a model that conforms to the ATL metamodel. Furthermore, the ATL metamodel conforms to the EMF metamodel.

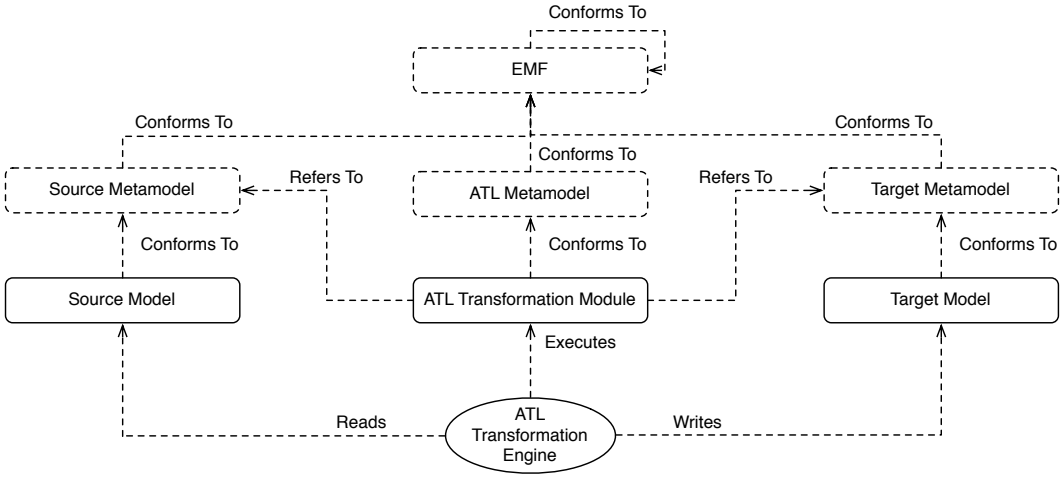


Figure 2.5: ATL Schema

Because of the model nature of the ATL modules, it supports *Higher Order Transformations* (HOT). A higher-order transformation is a model transformation whose input and/or output models are themselves transformation models [TCJ10]. This kind of transformations are essential in our proposal in order to be able to automatically generate new model-to-model transformations that support the evolution of an existing MTC. An MTC is a set of transformations that is put together to convert a high-level model, which is rooted in the problem domain, into a low-level model that is rooted in the solution domain. The notions behind an MTC are presented in the next subsection.

2.2.5 Model Transformation Chains

MDA promotes the specification of a software system in a high-level model (PIM). The PIM is specified only in terms of problem domain concepts and without any technological platform information. A model transformation takes the PIM as input and produces a low-level model (PSM) where the software system is specified in terms of the technological platform. However, the use of a single "almighty" model transformation to produce applications is impractical and almost impossible to maintain and evolve [VVBH+06]. Having

a single PIM-to-PSM transformation goes against the notions of abstraction and modularity. Instead, in order to tackle complexity in the generation of an application from a high-level model, it is necessary to split the transformation in several steps. Each of the transformation steps should be in charge of adding details to the transformed model.

A *Model Transformation Chain* (MTC) [PVSGB08, KGZ09, VVBH+06] is a set of transformations that are put together to convert a high-level model, which is rooted in the problem domain, into a low-level model that is rooted in the solution domain. In addition to the translation from problem domain concepts (e.g., Entity, Service) to solution domain concepts (e.g., Java Class, Java Annotation), the transformation chain adds implementation details in every transformation step. In the last step of the chain, there is a model-to-text transformation that produces the code of the software system. In this research we propose an approach for evolving MTC's.

Using this multi-step strategy allows developers to focus on one level of abstraction at a time and to reuse small transformations. There is not a fixed number of transformation steps in an MTC. The number of steps depends on the generation strategy taken, which depends on factors such as: concerns, stakeholders, transformation reusing, etc.

A possible MTC is presented in Figure 2.6. In the figure, the MTC artifacts (i.e., metamodels, models, and transformations) are presented as well. In this particular MTC, four transformation steps are combined in order to generate applications. This MTC takes a high-level model that conforms to a *Business metamodel* and transforms it in several steps into a low-level model that conforms to a *Java metamodel*. Finally, this model is transformed into Java code.

When it is necessary to analyze the impact of a change in the source models, to propagate changes, or to verify if the requirements are fulfilled by the code, *tracing models* are required. These models are generated when the model MTCs are executed and they identify the source elements and their corresponding target elements through the whole set of transformations. We discuss this in the following subsection.

2.2.6 Model Traceability

In MDE, when tracing information is required outside the scope of the execution, it is stored in the form of *tracing models*. These models represent the relationships between source elements and the target elements generated by the transformation. There are several uses for these models [ARNRSG06]. For instance, to analyze the impact of a change in the source models, to propagate the changes, to verify if the requirements are fulfilled in the code, or as input in complex transformation chains. [VBJB]. In our approach, traceability models are a fundamental element to evolve an existing MTC.

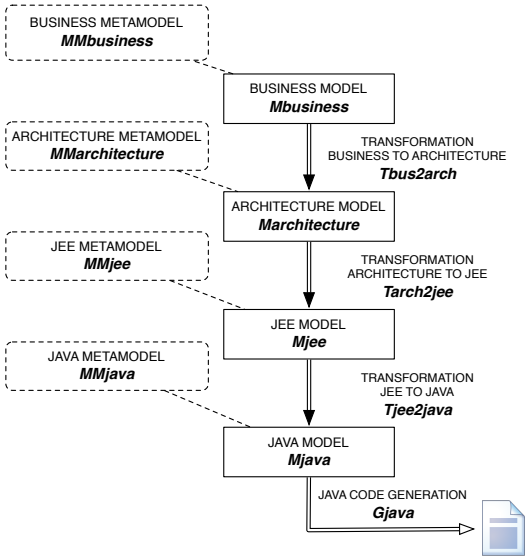


Figure 2.6: A possible instance of an MTC

Tracing models must conform to a tracing metamodel. A simple tracing metamodel is presented in Figure 2.7. This metamodel has two metaclasses: *TraceLink* and *TracedElement*. The first metaclass represents a tracing link that relates a set of source elements with a set of target elements. The second metaclass represents any element that is used by the transformation as a source or any element that is generated by the transformation.

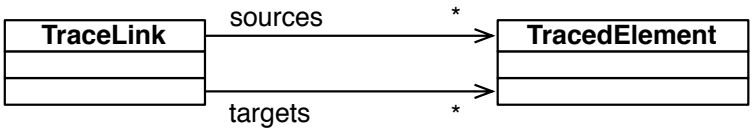


Figure 2.7: Basic Traceability Metamodel

This metamodel can be extended in order to have multiple types of trace links and to store additional tracing information. A tracing model richer in semantics and in traceability information can be used as input for subsequent transformations. For instance, the work by Vanhooff et al. [VBJB] presents a Global Traceability Graph (GTG) which is used as a mechanism to navigate several models involved in a transformation chain. These models represent different aspects of a system. The GTG is generated by the MTC. Sometimes, in order to produce a new model, it is necessary to know something from a different model in the transformation chain (e.g. to know the database table that stores a particular attribute from the functional model). In this approach, a transformation chain always starts from a single root model, and produces different kinds of models (e.g. application model, GUI model, database model)

by using several transformation branches. With the GTC it is possible to navigate every single model produced by the transformation chain.

There are several options to generate tracing models:

1. To extend the existing transformation rules with the tracing generation logic [FHN06].

2. To use a high-order transformation that automatically adds the tracing generation logic to the existing transformation rules [Jou05].

3. To use tracing provided by the transformation engine or tracing constructs of the language [JK06].

In our research we choose the last option and extend the ATL virtual machine implementation to automatically generate tracing models every time that a transformation is executed. The objective of this extension is to improve the usability of our approach and to give access to the traceability information for the ATL users. This extension is presented in Section 6.5.

In the previous sections we introduced the notions behind the use of models as prime artifacts to develop software in MDE. Also, we explained how models are specified by using a DSML, which is defined in a metamodel. Finally, we presented how models are processed by several transformations, which are put together in form of an MTC.

Although MDE offers advantages to the software development process in terms of increasing the level of abstraction, as every software system, MDE implementations should evolve. In the specific case of MDE, its evolution poses several problems related to the strong dependencies existing between its artifacts. Specifically, the problems caused by the evolution of metamodels and transformations are extremely challenging.

2.3 Problem: Evolution in Model Driven Engineering

Evolution is an inherent characteristic of software systems. For instance, to include new functionality, to address new non-functional properties, or to migrate to a new technological platform. Similarly, MDE implementations are also susceptible to evolution. Even though these MDE implementations are a considerable improvement allowing better platform independence and stability of the application models, one is confronted with a number of problems if models, metamodels, or transformations need to evolve. Evolution of MDE confronts several problems mainly related to the strong dependencies between metamodels and models, metamodels and transformations, and between each

transformation step and those which follow it. In [VWDD07], Van Deursen et al. divide MDE evolution into four categories:

- *Regular evolution:* this type of evolution affects the existing models when the specified application needs new requirements and is managed by the capabilities and abstractions of the modeling language. Hence, the models are modified but metamodels and transformations remain fixed. MDE is optimized for managing this type of evolution due to the higher level of abstraction promised by the modeling approach.

- *Metamodel evolution:* this type of evolution is required when the language needs to evolve and it has a huge impact on the existing models that conform to the original metamodel. Usually, the existing models need to be migrated. This is an important research field in the MDE community and is commonly know as *metamodel and model coevolution* problem [Wac07, Fav03, CDREP, GJCB09].

- *Platform evolution:* this type of evolution is when new requirements related to the target platform are added. When this happens, the model-to-model and the model-to-text transformation must be modified. The existing models and metamodels remain fixed. A research that focuses on this type of evolution in presented in [RB08].

- *Abstraction evolution:* this type of evolution appears when new modeling languages are added to and existing set of (modeling) languages to reflect an increased understanding of a technical or a business domain. This requires the modification of models, metamodels and transformations. A research that focuses on this type of evolution in presented in [MV09]. In this dissertation a solution to this type of evolution is presented as well.

In the next subsections we present a research related to the main problems that MDE evolution needs to tackle. As we explained before, these problems are caused by the strong dependencies that exist between metamodels and models, metamodels and transformations, and between each transformation step and those which follow it. We group these problems based on the artifact that is evolved (i.e., metamodels and transformations). Finally, we discuss the problem that we call *ripple effect*, which is the necessity of changing several artifacts in an MTC as a consequence of a single change in one of the metamodels or transformations.

2.3.1 Metamodel Evolution

When the domain evolves, then its metamodel needs to evolve as well. The evolution of a metamodel means that new concepts could be added or existing

concepts could be modified or eliminated. If concepts are added to a metamodel, then models are still conforming to the evolved metamodel. However, if the existing concepts are modified or eliminated, then the existing models may not conform to the new version of the metamodel. Therefore, to fix the problem, every model that conforms to the evolved metamodel must be modified to correct the inconsistencies. For instance, if an attribute of a metaclass is eliminated, it will be necessary to fix the models eliminating the attribute from them. If the models are not co-evolved with the metamodel, then they become invalid. This is known as metamodel and model coevolution, which is addressed by several researches [Wac07, Fav03, CDREP]. Additionally, when the metamodel is adapted, a transformation that uses it as a source or target must be adapted as well. The problem of transformation evolution is presented in Subsection 2.3.2.

In [RHW+09], Rose et al. divide the approaches to coevolve a metamodel and the models that conform to it into three groups:

- The first group is called *manual specification* and the idea is to manually develop a migration strategy in a programming or a transformation language. This strategy requires a huge effort from the metamodel developer.
- The second group is called *operator based coevolution*. In this group the metamodel is evolved by using a set of operators that are provided. Each one of these operators is specified with a migration strategy to be applied to the models that need to be migrated.
- Finally, the last group is called *metamodel matching*, in which a difference model is calculated by differentiating the old and new metamodels or by recording the changes to the existing metamodel.

In the group of *manual specification*, models are adapted manually in order to evolve them. Manually performing this adaptation is an error-prone task that could introduce inconsistencies between models and metamodels. The ideal is to automatically update the models in order to avoid inconsistencies.

In the second group we find the work of Wachsmuth [Wac07]. In this work the author proposes a transformational approach to assist metamodel evolution by stepwise adaptation. This means that the metamodel is evolved by using transformations that abstract common operations such as refactoring, constructing or destructing metamodel elements. This approach offers several advantages over manual *ad-hoc* adaptation: 1) changes become explicit and can be used as means of documentation and traceability, 2) several preservation properties of transformations are stated that allow for qualifying an adaptation according to semantics or instance-preservation, 3) coadaptation of models is achieved automatically by co-transformations, and 4) adaptation

scripts are pieces of software on their own that can be reused in similar adaptation scenarios or can be modified to alter adaptation decisions.

Finally in the last group there are works like the one presented by Cicchetti et al. in [CDREP, GJCB09]. In this work, it is proposed that the problem of metamodel evolution is tackled in a high level way by using a difference metamodel together with the related change representation mechanism. In particular, the proposed automation is based on a model difference representation, which is used to record the metamodel changes in a difference model. Thus, the coevolution is performed by using a HOT which takes the difference model and produces a model transformation able to coevolve the existing models.

When a metamodel is adapted, the transformations that use the metamodel as source or target must be modified as well. In the next subsection, we will present researches that tackle the problems of *metamodel and transformation coevolution.*

2.3.2 Transformation Evolution

Similar to metamodels, transformations are evolved due to the necessity of supporting new domain concepts, new platforms, etc. The situations where model transformations must be evolved are:

1. When the source or target metamodels are modified and the transformations must be adapted to support the new or modified concepts.

2. When the technological platform evolves or new requirements related with the platform are added.

3. When new languages are added to the implementation and the new concepts need to be integrated in the model transformations.

A research work that offers a solution to model transformation evolution is presented by Roser and Bauer in [RB08]. In this work the authors propose an approach of ontology-based model transformation. This approach integrates ontologies to automate the generation and evolution of model transformations. Their proposal facilitates methods to generate and adjust model transformations despite of the structural and semantic differences of metamodels. Different representation formats and different semantics are overcome by annotating metamodels using concepts of a reference ontology. The reasoning over the annotated metamodels allows for generating and adjusting common model transformations automatically.

There are not many works related with the evolution of model transformations, mainly the evolution is managed by source evolution techniques or with model transformation composition techniques. In model transformation

composition techniques, there are several works such as the work presented by Vanhoof et al. in [VB05], in which traceability information is injected and used by transformation steps that are composed one after the other. The work presented by Wagelaar et al. in [WVDSD10] propose a superimposition technique for composing rule-based model transformation languages. Finally, the work presented by Etien et al. in [EMLB10] propose a mechanism to reuse transformations in MTCs. The idea of this mechanism is to extend an existing transformation by obtaining a new transformation that has as source metamodel an extended version of the original source metamodel.

However, when a metamodel or a transformation of a MTC is adapted, it will trigger a series of changes in the metamodels and transformations that follow the adapted artifact.

2.3.3 Ripple effect

As mentioned before, there are strong dependencies between the MTC artifacts. These dependencies are between metamodels and models, metamodels and transformations, and between each transformation step and those which follow it. In other words, when an MTC metamodel or transformation is modified due to the reasons mentioned before (e.g., domain evolution, platform evolution), the subsequent metamodels and transformations must be adapted as well in order to support the new domain concepts, or the new platform. Therefore, if a change impacts a metamodel or a transformation, then the complexity and cost of maintaining the consistency among all the artifacts will increase. Therefore, every step should be carefully adapted to assure that the chain is not broken and does indeed support the new concepts [KKS07].

In order to avoid directly modifying the existing metamodels and transformations and to avoid the problems presented in Section 1.1, we envision a solution that allows us to add the new concern independently of the existing MTC. A strategy that can be used to maintain the existing MTC oblivious of the change is to follow the SoC principle in the evolution of an existing MTC. In order to understand how we apply the SoC principle in our solution, we introduce the SoC ideas applied into MDE in the next section.

2.4 Solution: Separation of Concerns in MDE

There are some MDE approaches that follow the *Separation of Concerns* (SoC) principle [Dij82] which states that the different aspects of a system should be specified separately. In this section we present the two main approaches (i.e., AOM and multi-modeling) from which we take some ideas for our solution to evolve an existing MTC. We use these ideas to specify the new concerns independently of the existing MTC artifacts. Additionally, in this section we

present the notions behind model composition that we use in order to include the new concern in the generated application.

2.4.1 Modeling Concerns

One of the approaches that follows the SoC principle is Aspect Oriented Modeling (AOM)[SSK+07], which is based on the ideas of *Aspect Oriented Software Development* (AOSD). This approach aims to provide new ways of modularization in order to separate crosscutting concerns from traditional units of decomposition during software development. Another approach that follows the SoC principle and goes further is *Multi-modeling* [LH09]. In Multi-modeling, multiple Domain Specific Modeling Languages (DSMLs) are used to specify multiple concern models in order to specify a complex system. Each DSML provides the most appropriate abstractions for each concern involved.

Aspect Oriented Modeling

The idea of encapsulating crosscutting concerns separately from the main program in stand-alone modules was applied first by Aspect Oriented Programming (AOP) [KLM+97]. The main goal of AOP is to offer developers a way to specify crosscutting code in separated modules. These modules are called *aspects* that contain the *advice*. The advice is the code that is going to be inserted with the *base* program. The execution point where the advice is inserted is called a *join point*. The possible join points are specified (quantified) by a *pointcut* expression. In AOP the composition of advices with the base program is called weaving. Finally, AOP languages usually use a syntax similar to the base language that improves their usability and homogeneity. An example of an AOP language is *ApectJ* which uses Java signatures as pointcut expressions.

With the arrival of model based techniques, AOP principles were translated to earlier phases in the software development cycle such as requirements, analysis and design [SSK+07]. The goal of *Aspect Oriented Modeling* (AOM) approaches is to specify a complex system by modeling multiple concerns at a high-level of abstraction and to offer a mechanism to integrate the different models.

It is possible to divide AOM approaches into two main categories [FR07]. On the one hand, there are approaches that provide modeling abstractions for AOP concepts. Works in this category allow for modeling concepts such as join points and advices to use extensions to UML. Once the main program and its aspects are modeled, it is possible to generate AOP code using transformations. On the other hand, there are approaches that offer modeling techniques for the requirements, analysis and design phases that allow us to specify multiple concern models separately and to analyze interactions across the concern

models.

As presented before, in the first category, concepts such as join points and advices are modeled [JCC09, Nai09]. It should be noted that these concepts belong to the solution domain and not to the problem domain. As a result, this situation creates a dissonance between the MDE goal of leaving out platform concepts at high-level of abstraction and the AOP concepts used at a high-level of abstraction. In this category, the models are translated to AOP code and composition of the different aspects is performed by the weaving mechanism.

In the second category, multiple concerns are specified in the early phases of the software development cycle [GS09, MBC09, BCA+06]. Traditionally, AOSD has focused on the implementation phase: aspects are identified and captured mainly in code. However, aspects are evident earlier in the life cycle, such as during requirements gathering and architecture development. Identifying these *early aspects* ensures that you can appropriately capture aspects related to the problem domain. Additionally, it offers opportunities for early recognition and negotiation of trade-offs and allows forward and backward aspect traceability. This makes requirements, architecture, and implementation more seamless and lets you apply aspects more systematically. In the field of AOM, the early aspects are defined as models are specified at high-level of abstraction. At a high-level of abstraction, each model represents a concern in its early stages and is specified using concepts that belong to the problem domain. In order to analyze the interactions among multiple concerns or to generate the full application, these approaches must offer a composition mechanism. Usually, the composition mechanism is based on name matching techniques to identify the elements to compose.

A commonality between both categories is the use of a common modeling language, such as UML. In the first category the common modeling language is extended to specify join points and advices, and in the second category it is extended to specify the different application concerns. In the case of the first category, models are translated to the code level where they can be composed using an AOP weaving mechanism. In the second category, the elements are integrated based on syntactical similarities (e.g., model element name) and take advantage of the common modeling language used to specify the base model and its concerns. In both cases the use of a common language to specify the base model and the different concerns avoids the complexity of integrating models specified using heterogeneous languages. This heterogeneity would impose the definition of a semantic alignment.

An additional approach that follows the SoC principle and uses multiple DSML to specify various concerns is called *Multi-modeling*. Multi-modeling is presented in the next subsection.

Multi-modeling

As presented before, DSMLs are created to reduce the gap between the problem and the solution domains through the use of domain abstractions. However, when a complex system is specified, it usually does not fit cleanly in a single domain. Additionally, following the SoC principle, if it is necessary to specify several concerns closer to the problem domain, then it is necessary to use multiple DSMLs [HCW07]. These approaches are grouped in *multi-modeling* techniques. In these approaches, the system should be specified using the most appropriate modeling language(s) at the most appropriate level(s) of abstraction [MV02].

An example of a multi-modeling approach is presented by Cicchetti and Di Ruscio in [CDR08]. In this work the authors present a proposal for modeling a web application using three independent models: a data model, a composition model, and a navigation model. Each of these models is expressed with an appropriate DSML, such as a Presentation DSML that expresses the presentation model using concepts of the presentation domain, such as *dialogs*, *fields*, *buttons*, etc.

Multi-modeling approaches use several models which are specified using different DSMLs in oder to specify a complex system. Consequently, each model conforms to a different metamodel. Although the use of different metamodels helps to specify the systems with concepts that are close to the problem domain, it increases the complexity of combining, merging, and integrating models that are specified at different levels of abstraction and conform to different metamodels. Additionally, *Architectural Description Languages* propose the use of several views to specify a complex system. These languages are presented in the next subsection.

Architecture Description Languages (ADLs)

As the size and complexity of software systems grows, there is a increasing need to describe the underlying structures of a system. The *Architectural Description* (AD) is used to specify these structures as a collection of architectural models grouped into views. An *Architectural Description Language* (ADL) defines the language concepts and semantics in which an AD is expressed [MT00]. Early ADLs typically provide a language to describe a single concern of an architecture [MDT07]. Later ADLs provide broader support, and try to build a language that can be extended to support several concerns. Among these extended ADLs we can find xADL. xADL is conceived as a language for constructing domain specific ADLs. [DHT05].

When an AD consists of multiple models, an architect faces the problem of bringing these models back together. In other words, a key requirement when using multiple models is knowing how the individual models are related

and knowing how the information in the models can be integrated to get a better understanding of the architecture as a whole. Furthermore, the relationships that can be defined among the multiple AD models can be used for several purposes, such as: composition [BH08, DRMM+10], consistency checking [NEFE03] and achieving consistent changes through the multiple models [THA07].

In [BH08], Boucke et al. propose an extension to xADL that offers three new types of relationships that allow specifying and composing structural views. These relationships are:

Unification: Unifies elements from different structural views with each other. In other words it expresses that elements that appear in the different views are actually the same element.

Mapping: Maps individual elements or groups of elements (called subjects) from one structural view on a single element of another structural view (called target). The subjects then become subelements of the target element.

Refinement: Defines that a specific structural view (referred to as inner structure) describes a substructure for an architectural component (referred to as outer component) of another structural view.

These relationships are first-class in the AD and supported by the ADL. Additionally, they implemented a composition mechanism that integrates the multiple views in a single model.

In [DRMM+10] Di Rucio et al. proposes a framework, called `BY`ADL for developing new generation of ADLs. This framework uses model-driven techniques that allow a software architect, to define its own new generation ADL by 1) adding domain specificities, new architectural views, or analysis aspects, 2) integrating ADLs with development processes and methodologies, and 3) customizing ADLs by fine tuning them. Additionally, `BY`ADL proposes four types of composition operators:

Match: the related metaclasses semantically overlap and then they are merged into a single metaclass. The composed metaclass contains the union of all the structural features of the related metaclasses.

Inherit: this operator specifies that one related metaclass will be a subtype of another related metaclass in the resulting composed metamodel.

Reference: In the composed metamodel one related metaclass references to the other related metaclass.

Expand: The attributes of one related metaclass are copied into the other related metaclass. It is possible to specify if the source is copied to the composed metamodel.

One of the differences between the Di Rucio work [DRMM+10] and the Boucke work [BH08] is that in the former the composition relationships are independently defined at metamodel level between two different metamodels in order to interoperate multiple ADLs. In the latest the relationships are integrated in a fixed ADL to integrate its different views. The Di Rucio work allows a more flexible solution to integrate models that are specified using different languages. In our work we choose a similar approach as Di Rucio independently defining our composition operators at metamodel level.

In the next subsection we will explain the concepts behind model composition.

2.4.2 Composing Concerns

When several models are used to define a complex system, these models must be integrated in order to obtain the required application. Each one of these models represents a concern of the complex system and can be specified using a common modeling language or multiple DSMLs. In order to obtain the required application we will need to perform a *model composition*.

Model composition is an operation that integrates two or more models into a single model [DFBV06]. This composition is performed by using a model-to-model transformation that requires two or more models as input and integrates their elements into a single output model.

The most basic requirement to be able to compose two models is to identify which elements will be composed [JFB08]. Most compositional approaches are based on the use of the signature of the elements or a syntactical matching algorithm to identify if two model elements represent the same concept or not. However, it is not always possible to rely on syntactical properties and it is necessary to use semantic properties than can be expressed as constraints or as specifications of behavior when matching model elements [FR07].

When it is not possible to use syntactical or semantical properties to automatically identify the elements to compose, the only possible solution is to manually identify the elements that represent the same concept. This identification can be done by establishing correspondence relationships between the elements to compose. These relationships can be defined in a *Correspondence Model (CM)* that contains the links between the models to compose [BBDF+06].

Correspondence Models & AMW

In [BBDF+06], Bézivin et al. define a *Correspondence Model* as a model representing links between elements of different models. The metamodel of the correspondence model (correspondence metamodel) needs to be extensible, because different matching and composition links can be defined (e.g., match, override, correspondence, equality, merge, joint). The main elements in the correspondence metamodel are: *link* and *link endpoint*. A *link* represents a correspondence relationship between two elements in two different models. A *link endpoint* refers to an element in one of the models by means of identification functions. An identification function takes a link endpoint as input and returns an element of a model.

A current implementation of a correspondence model is the *Atlas Model Weaver* (AMW) [DFBV06]. AMW is a model composition framework that uses model weaving and model transformations to produce and execute composition operations. AMW is used to define relationships (i.e., links) between models. The links are stored in a model, called *weaving model* that conforms to a weaving metamodel.

In Figure 2.8 the weaving metamodel provided by AMW is presented. This metamodel allows the creation of links between model elements and can be extended to add other correspondence semantics. The main elements are:

1. *WModel* that represents the root element that contains all model elements.
2. *WLink* that expresses a link between model elements.
3. *WLinkEnd* that represents a linked model element.

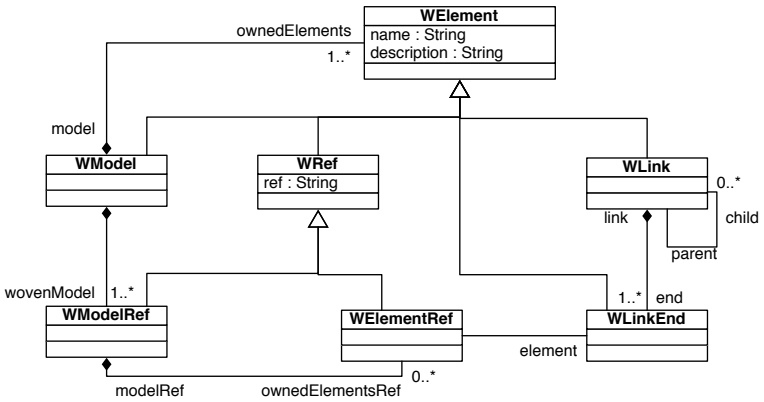


Figure 2.8: AMW Metamodel (source [DFBV06])

In our research, we took inspiration from the AMW metamodel and plugin. We did not use AMW because of compatibility and stability problems with the

new versions of Eclipse and ATL that were used in our research. The AMW implementation was not maintained during the time of this research. For this reason, we chose to define the *Correspondence Metamodel*. This metamodel is presented in Section 5.2. Additionally a modeling tool was implemented to allow creating instances of this metamodel. This tool is presented in Section 6.4.1.

Once the corresponding elements in two models are identified, a composition of both models must be performed. In the next section we will explain how model composition is performed.

On the one hand, when two or more models are expressed in a single DSML (i.e., the models conform to the same metamodel) a *homogeneous composition* is performed. On the other hand, when the models are expressed in different DSMLs a problem arises because a *heterogeneous composition* is required. Next, we will explain both types of composition.

Homogeneous composition

A *homogeneous composition* is the integration of models that conform to the same metamodel. This means that the composition is performed between elements that conform to the same metaclass (e.g., composition of two Classes). This is an ideal situation because there is no need to semantically align the concepts of the language (i.e., they are the same concepts), and a single composition mechanism can be used. Additionally, if the two models conform to a GPL based metamodel it is possible to go a little further and take advantage of existing GPL based composition mechanisms.

In summary, the advantage of a homogeneous composition it that it is possible to use the same composition mechanism for any number of concern models because every model conforms the same metamodel. Additionally, if we transform the multiple models into actual code, we can use code-based composition mechanisms such as AOP.

Heterogeneous composition

Sometimes a system is specified using two or more models using several DSMLs that are close to the problem domain. When this happens a *heterogeneous composition* must be performed.

A *heterogeneous composition* is the integration of two models expressed in two different languages. This means that the compositional semantics for every pair of concepts in both languages needs to be defined for each pair of languages to compose. For instance to compose a Business model that conforms to a Business metamodel and a Security model that conforms to a Security metamodel a heterogenous composition will be required. For example, if we have the concept *Entity* that belongs to a Business modeling language, and

the concept *Resource* that belongs to a Security modeling language, then we have to define what it means to compose them. For example, if the composition means that the *Entity* is protected, then every *Service* and *Attribute* of the *Entity* will be protected as well. Similar, if a third language is added, a composition semantics for each pair of languages will be required, increasing the complexity of implementing a multi-modeling solution.

If the involved DSLs are fixed then it is possible to define a composition mechanism that is capable of integrating the models expressed using the defined languages. For instance, in the work that we mentioned before by Cicchetti and Di Ruscio in [CDR08], a web application is modeled using three independent models: a data model, a page structure model, and a navigation model. Each of these models is expressed with an appropriate DSML, for example a Presentation DSML expresses the presentation model using concepts of the presentation domain, such as *dialogs*, *fields*, *buttons*, etc. In this proposal, the three models are the inputs of a transformation rule that composes them and generates an integrated model that conforms to a common target metamodel. In this case the three input languages are fixed and implementing a composition mechanism for them is a suitable option.

However, if a new DSML (e.g., a Security DSML) is added to the existing three (i.e., Data DSML, Page Structure DSML and Navigation DSML), then an adaptation of the composition transformation will be required. This adaptation should manage the concepts of the new DSML as well as integrate the new models specified with the new DSML with the other three models. Therefore, for any unanticipated added DSML, the composition mechanism must be modified increasing the costs of maintenance of the MDE implementation.

In [VdL02], Vangheluwe et al. state that in order to simulate a system that is specified in different formalisms (e.g., Petri Nets, Finite State Automata, State Charts, Queueing networks) there are three options:

1. Using a *super-formalism* which subsumes the different formalisms of the sub-models (e.g., DEVS&DESS Zeigler et al., 2000)

2. Transforming the different sub-models into one common formalism.

3. Using a co-simulation approach, where each of the sub-models is simulated with a formalism-specific simulator.

In our work, in order to reuse a low-level composition mechanism and to reduce the complexity of evolving an existing MTC, the strategy that fits best is transforming each model into a common formalism (i.e., a common metamodel in our case). Using this strategy, we avoid having to perform a *heterogeneous composition* at high-level of abstraction and having to perform a *homogeneous composition* at the low-level of abstraction. Moreover, if the common metamodel is a technological platform metamodel, it is possible to take advantage

of an existing composition mechanism for the platform. Usually, at platform level there are several tested and commonly used composition mechanisms, such as AspectJ for Java.

2.5 Summary

In this chapter we have presented the concepts and ideas behind Model Driven Engineering (MDE), several works that try to tackle the problems that arise when an MDE is evolved, and the concepts behind the application of the SoC principle to MDE, which are used in our solution.

In the first sections, we show how MDE helps to increase the level of abstraction using models, metamodels, and transformations. The models that are expressed at the high-level of abstraction are used to generate low-level implementations by using automatic model transformations. Several of these model transformations are chained together in order to produce applications. The use of several transformation steps helps to tackle the complexity of implementing a single "almighty" transformation. Additionally, we have explained the importance of model traceability when automatic transformations are used. Next, several works are presented showing the state of the art in MDE evolution. Finally, in addition to the leverage given by abstraction, when complex systems are specified, it is important to follow the SoC principle to modularize the system. We apply the SoC principle in our solution by using the concepts behind AOM and Multi-modeling techniques.

Chapter 3
Evolving a Model Transformation Chain

3.1 Introduction

As we search for a suitable strategy to evolve an existing Model Transformation Chain (MTC), we look for inspiration in other research fields such as Aspect Oriented Programming (AOP) [KLM+97], Aspect Oriented Modeling (AOM) [SSK+07], Multi-modeling [LH09], etc. We have studied multiple works seeking out strategies that allow the specification of several concerns and which are automatically composed to produce a full application. We translate the ideas of these works into the field of MTC evolution and we analyze the strategies in order to identify advantages and disadvantages of each one. We define the key criteria that allow us to compare the strategies. These criteria allow us to select the most suitable strategy according to our goal of performing a non-invasive evolution of an existing MTC.

In order to explain the selected strategies, we first introduce a case study in Section 3.2 where an instance of an MTC and its components (i.e., metamodels, models and transformations) are illustrated. In Section 3.3, we present an evolution scenario where a new concern is added which cannot be expressed using the existing MTC's metamodels. Additionally, in this section we describe the key criteria that we use to compare the potential strategies. Finally, in Section 3.4 we present the analysis and comparison of the different strategies using the key criteria.

3.2 A running example: Business-to-Java MTC

As presented in Section 2.2.5, an MTC is a set of transformations that are put together in order to convert a high-level model that is rooted in the problem domain into a low-level model that is rooted in the solution domain. In addition to the translation from problem domain concepts to solution domain concepts, the transformation chain adds implementation details in every transformation

step. In the last step of the chain, there is a model-to-text transformation that produces the code of the software system.

In order to illustrate the problems that arise when a non-expected concern is added to an existing MTC, we will use a running example. For this, we use an instance of an MTC that allows specifying applications based on components and generating their implementation in Java Enterprise Edition (*JEE*) [CS09]. The MTC takes the high-level model that conforms to the *Business metamodel* and transforms it in several steps into a low-level model that conforms to the *Java metamodel*. Finally, this model is transformed into Java code. The generated applications will offer basic CRUD functionalities such as insert, update, delete, and retrieve for the information of the business entities. This particular case study is used throughout this dissertation for illustration purposes. Figure 3.1 presents the MTC with its metamodels, models, and transformations. We use this MTC to produce a risk management application named `Risk`. This application needs to manage multiple software projects that are developed in a company and the risks that affect each project (e.g., personnel shortage, missing requirements, schedule slips). A project is threatened by a set of risks and each risk has associated some mitigation plans. These plans allow the reduction of the impact of each threat in the project.

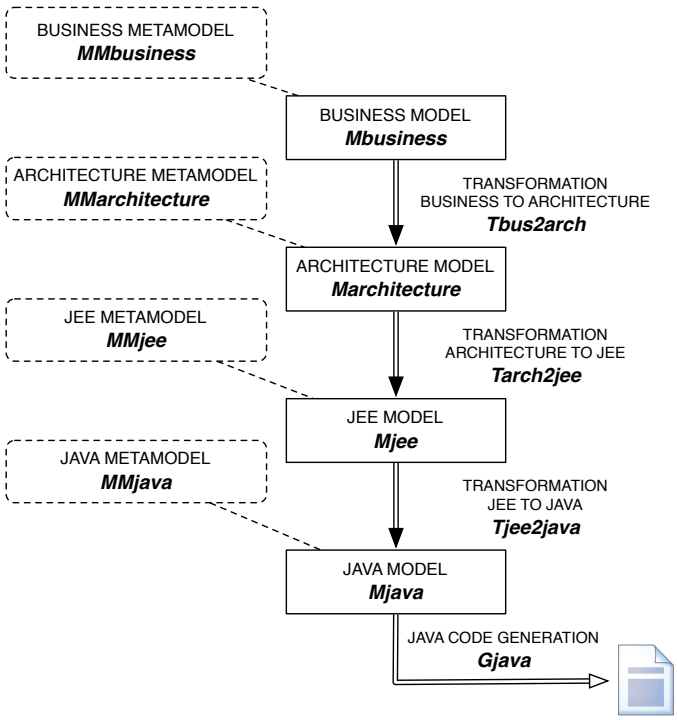


Figure 3.1: Business-to-Java MTC

Our case study uses a high-level metamodel in which the concepts of Enterprise Applications are represented. We call this metamodel the *Business*

Metamodel, which contains metaclasses such as *BusinessEntity* and *Service*. Using this metamodel we specify a *Business Model* for `Risk`. By applying several model-to-model transformations to the Business model, we obtain a low-level Java model that conforms to a Java based metamodel. We call this metamodel the *Java Metamodel* which contains metaclasses such as *Class*, *Field* and *Method*. Finally, this low-level model is transformed into Java code using a model-to-text transformation.

In this specific case study we decide to split the transformations from Business to Java into three model-to-model transformations and one model-to-text. The rationale behind this division is based on having four levels of abstraction related to the experts involved. The first level of abstraction is related to the business modeler. At this level the business modeler can specify the application using business concepts. Next, the first model-to-model transformation brings the business concepts into the level of software architecture concepts. These concepts are defined in the *Architecture Metamodel*. This metamodel contains concepts such as *Layers* and *Communication* between the layers. The next model-to-model transformation brings the architecture concepts into the JEE platform concepts. In this level, the architecture concepts are transformed into the actual JEE mechanisms. The used *JEE Metamodel* contains concepts such as *EntityBean* and *SessionBean*. The last model-to-model transformation transforms the *JEE model* into a *Java model*. This last step transforms the JEE concepts into actual Java concepts such as *Class* and *Method*. Finally, a model-to-text transformation transforms the Java model into text generating the application Java code and other required artifacts such as XML descriptors. This final transformation uses as input a model that conforms to a Java metamodel and serializes it in the form of Java files.

3.2.1 High-Level Business Metamodel ($MM_{business}$) and Model ($M_{business}$):

Figure 3.2(a) presents a part of the high-level business metamodel. The essential concept of this metamodel is *BusinessEntity*, which represents business entities of the application, such as `Project` or `Risk`. A *BusinessEntity* has *Attributes*, *Associations* and *Services*. Figure 3.2(b) shows a part of the `Risk` high-level model: `Project` represents a managed project; `Risk` represents a risk that threatens the `Project`. Additionally, each risk has mitigation plans represented by `Plan`. `Project`, `Risk` and `Plan` are concepts that conform to the *BusinessEntity* concept. In this model the *BusinessEntity* `Project` has the *Attributes* `name`, and `dueDate`. The *BusinessEntity* `Project` has the *Association* `risks` to the *BusinessEntity* `Risk` representing all the risks that threaten the project. Additionally, the *BusinessEntity* `Risk` has the *Association* `plans` to the *BusinessEntity* `Plan` representing the mitigation plans for a specific risk.

The complete metamodel is presented in Appendix A.1.

3.2.2 Architecture Metamodel ($MM_{architecture}$) and Model ($M_{architecture}$)

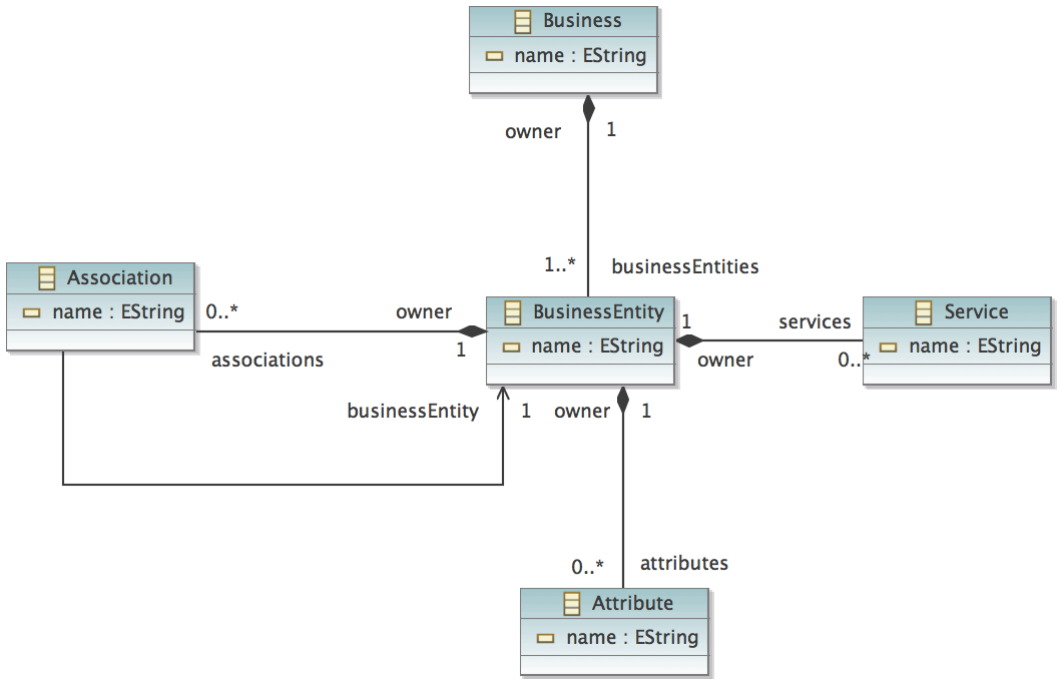
The main objective of the *Architecture Metamodel* is to provide concepts that allow the specification of a multi-tier application as well as platform independent architectural patterns such as the *BusinessDelegate* pattern. Subsequently, the main concept is *Layer*, which is an abstract concept specialized into *PresentationLayer, ApplicationLayer, SystemServicesLayer, BusinessServicesLayer, PersistenceLayer* and *DataSourceLayer*. These layers are related to each other by *Communications* that are constrained to only communicate "adjacent" layers. The constraints are expressed as OCL constraints in the metamodel. The complete metamodel is presented in Appendix A.2.

3.2.3 Business to Architecture Transformation ($T_{bus2arch}$)

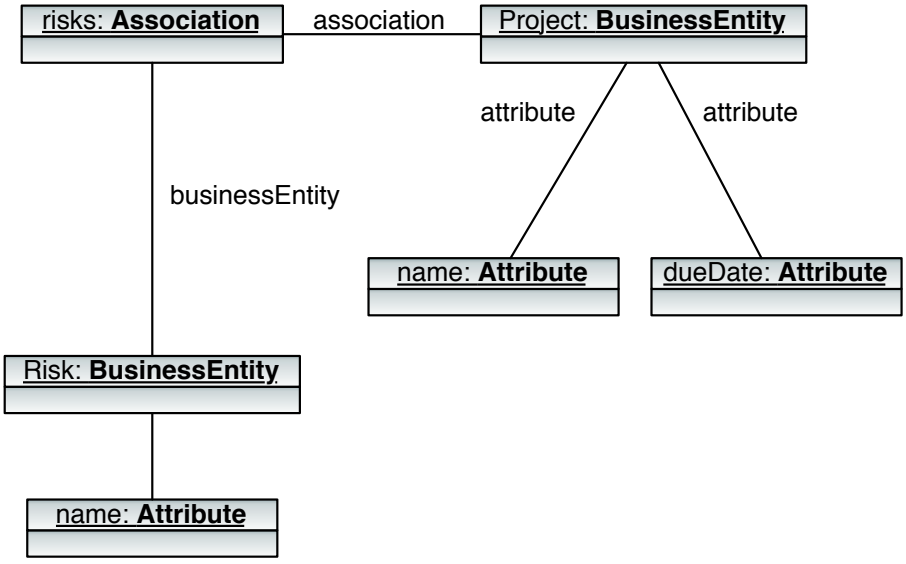
The *Business to Architecture* model-to-model transformation ($T_{bus2arch}$)[1] takes each *BusinessEntity* and transforms it into the multiple layers involved in the required multi-level application. The transformation is responsible for connecting each of these layers to the adjacent ones. Additionally, the transformation should connect the layers that came from a *BusinessEntity* with the layers that came from other *BusinessEntity* related to the first one. For instance, Figure 3.3 presents part of the transformation[2]. In this transformation the *BusinessEntities* `Project`, `Risk` and `Plan` are each transformed into six *Layers*. The *PresentationLayer* `Project` should be connected with the *ApplicationLayer* `Project`, the *ApplicationLayer* `Project` with the *SystemServicesLayer* `Project`, the *SystemServicesLayer* `Project` with the *BusinessServicesLayer* `Project` and so on. Additionally, the *SystemServicesLayer* `Project` should be connected to both the *BusinessServicesLayer* `Project` and *BusinessServicesLayer* `Risk` due to the relation `risks` between them. In Figure 3.3 only the transformation of the *BusinessEntity* `Project` is presented and even though for a simple element, one immediately sees the complexities involved in the corresponding transformation.

[1]The code of this transformation is available at http://qualdev.uniandes.edu.co/mtcframework

[2]The UML Package representation is used to depict a model.



(a) Business metamodel



(b) Business model

Figure 3.2: Business model and metamodel

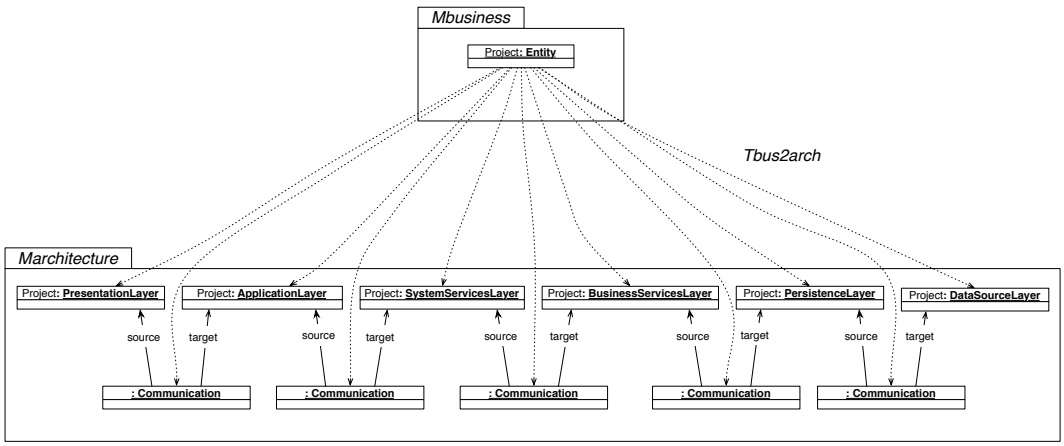


Figure 3.3: Business to Architecture Transformation ($T_{bus2arch}$)

3.2.4 Java Enterprise Edition Metamodel (MM_{jee}) and Model (M_{jee})

The main objective of the *Java Enterprise Edition Metamodel* is to provide concepts that belong to the JEE Application Server platform. This metamodel contains platform specific concepts of JEE such as *EntityBean* and *SessionBean*. These concepts are the mechanisms that implement the *SystemServicesLayer* and the *BusinessServicesLayer* in the JEE platform. The complete metamodel is presented in Appendix A.3.

3.2.5 Architecture to JEE Transformation ($T_{arch2jee}$)

The *Architecture to JEE* model-to-model transformation ($T_{arch2jee}$)[3] is in charge of translating the architectural concepts into actual technological platform concepts. For instance, the *SystemServicesLayer* is implemented in JEE as a *SessionBean*. Additionally, the *BusinessServicesLayer* is implemented in JEE as a *EntityBean*. At this level, the model represents the required application from the JEE expert point of view. Figure 3.4 shows the transformation for the *SystemServicesLayer* `Project` and `Risk` and for the *SystemBusinessLayer* `Project` and `Risk` into JEE concepts.

3.2.6 Low-Level Java Metamodel (MM_{java}) and Model (M_{java}) :

The main objective of the *Java Metamodel* is to provide a representation of the Java Language. This metamodel contains concepts such as *ClassDeclaration*,

[3]The code of this transformation is available at http://qualdev.uniandes.edu.co/mtcframework

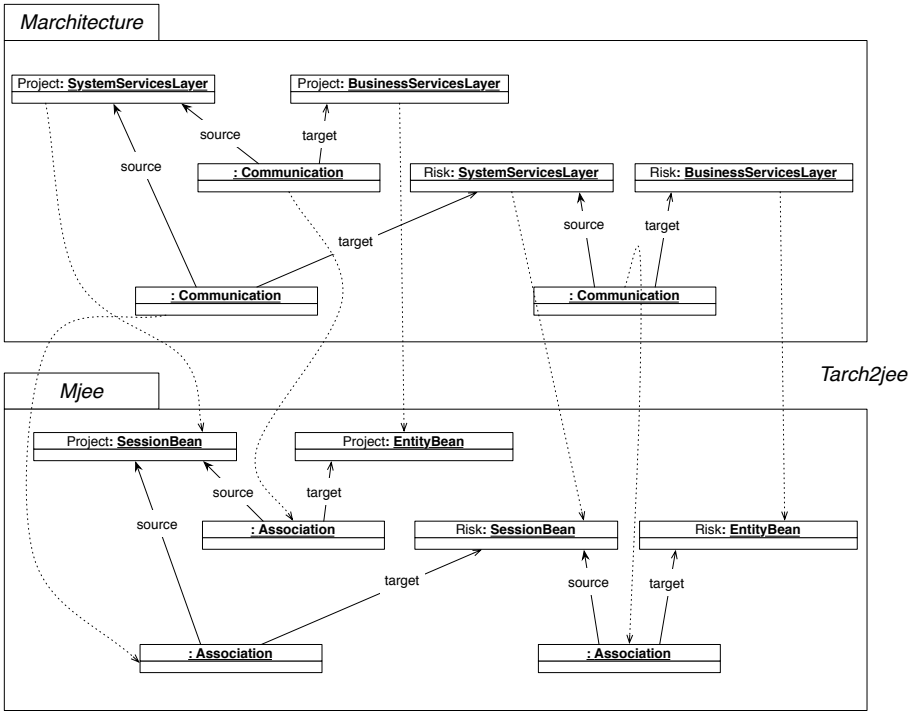


Figure 3.4: Architecture to JEE Transformation ($T_{arch2jee}$)

MethodDeclaration, *FieldDeclaration*, etc. At this level of abstraction every platform specific detail of the required application is specified in the model. The metamodel is based on the *J2SE5 Metamodel* of the MoDisco project (http://wiki.eclipse.org/MoDisco/J2SE5). This metamodel is the reflection of the Java language and a small fragment of it is presented in Figure 3.5. Additional fragments of the metamodel are presented in Appendix A.4.

3.2.7 JEE to Java Transformation ($T_{jee2java}$)

The final model-to-model transformation translates the JEE concepts into low-level Java concepts[4]. For instance, the *SessionBean* `Project` is transformed into the *ClassDeclaration* `ProjectSession` and the *InterfaceDeclarations* `IProjectLocal` and `IProjectRemote`. The *ClassDeclaration* implements both *InterfaceDeclaration*, which correspond to the actual Java implementation of a *SessionBean*. In addition, an *Attribute* is transformed into a *Field* and a getter and a setter *DeclareMethod* are added for it. Figure 3.6 presents part of the transformation.

[4]The code of this transformation is available at http://qualdev.uniandes.edu.co/mtcframework

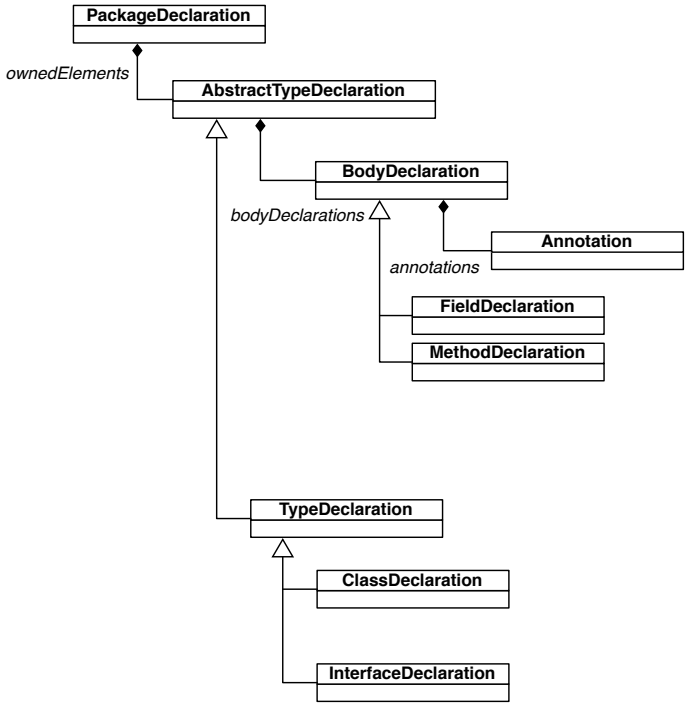


Figure 3.5: Java Metamodel (MM_{java})

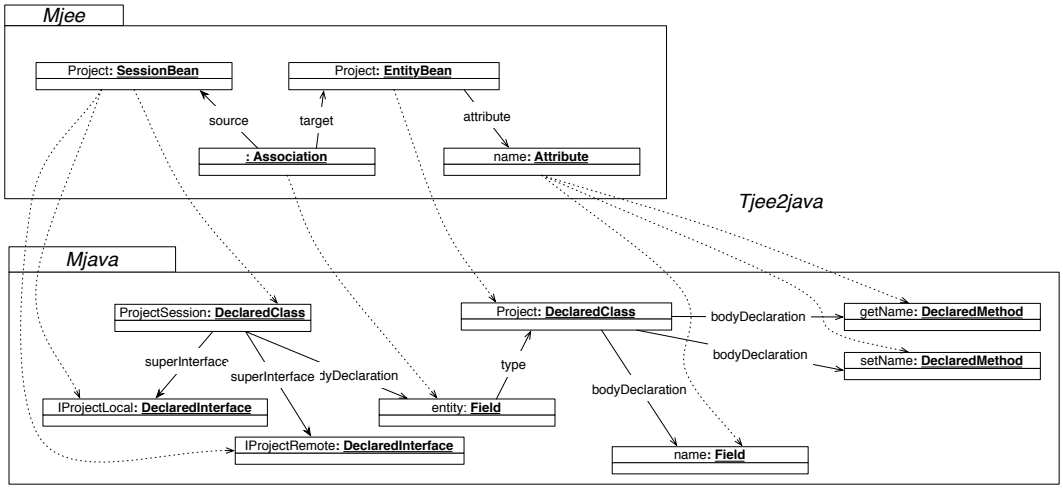


Figure 3.6: JEE to Java Transformation ($T_{jee2java}$)

3.2.8 Code Generation (G_{java}):

The last step is a model-to-text transformation that generates the Java files of the application. The code generation process takes every element in the model and serializes it in the form of Java files, method signatures and Java statements. These Java files are the platform specific representation of the required application. Moreover, XML descriptors are generated as well.

3.2.9 Business-to-Java MTC

The *Business-to-Java MTC* is used to produce a risk management Web application by transforming a high-level model into Java code. The high-level model is processed by three model-to-model transformations that translate the concepts of the Business model into Architecture concepts, then into JEE concepts, and finally into Java concepts. Each transformation translates the concepts and adds implementation details to the model. Finally, the Java code is generated by a model-to-code transformation. Figure 3.7 presents a screenshot of `Risk`. However, as the number of users of `Risk` increases, it becomes necessary to protect the application by adding authentication and authorization mechanisms. In the next section, we present an evolution scenario in which we will add a security concern to the existing MTC in order to generate secured Web applications.

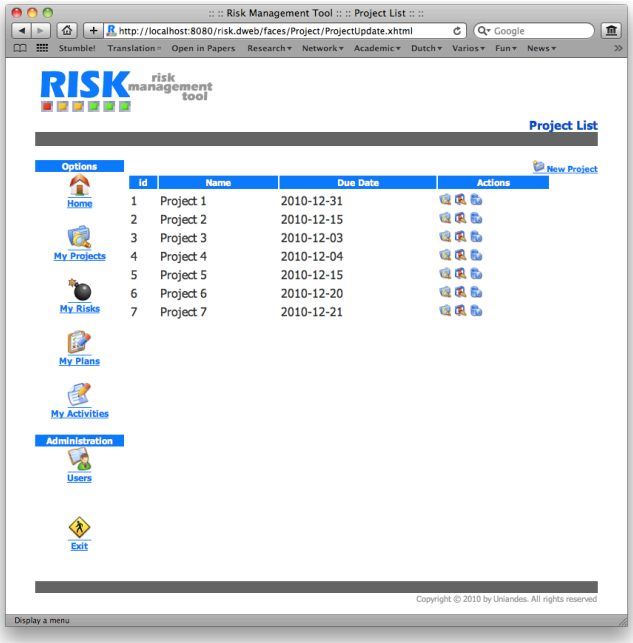


Figure 3.7: A Risk screenshot

3.3 Adding a new Concern

Suppose that the number of `Risk` users has increased and as a consequence it is now required to modify the MTC in order to generate a secured version of it. Specifically, we must control the users access to `Risk` information and services. The new authorization requirements for `Risk` cannot be specified and implemented using the existing MTC. In other words, the *Business metamodel* concepts cannot be used to specify security, neither the existing transformations. A possible solution is to directly extend the existing MTC. However, with this solution we will find several problems due to the impact of the changes in the MTC artifacts. These changes are necessary in order to include the security concepts and platform specific mechanisms to enforce the authorization policies.

In order to choose the best strategy to evolve the existing MTC, we look for inspiration in other research fields such as Aspect Oriented Programming (AOP) [KLM+97], Aspect Oriented Modeling (AOM) [SSK+07], Multi-modeling [LH09], etc. We choose works where a concern was independently specified from the main application and automatically composed with it. Inspired by these works we identify five possible strategies that can be used to evolve an existing MTC and we compare them using the key criteria. From the analysis of the different strategies we identify the most suitable one and adapt it to evolve an existing MTC.

3.3.1 Key Criteria

When it is necessary to modify an existing MTC by adding a new set of requirements that cannot be specified using the existing assets, it is important to use an approach that preserves the original MTC artifacts. We have identified the key criteria as a basis to compare potential strategies and choose the most suitable one to evolve an existing MTC. Each key criterion is related with one or more of the research goals presented in Section 1.2. Additionally, each key criterion is paired with a question that allows us to determine if the strategy has a positive or a negative impact on the evolution of an MTC. A summary of the research goals that were presented in Section 1.2.2 is showed in Table 3.1 and a summary of the key criteria and their respective questions is presented in Table 3.2.

Criterion 1 (C1): Impacted artifacts

As previously explained, there are strong dependencies between the MTC artifacts. These dependencies are between metamodels and models, metamodels and transformations, and each transformation step and those that follow it. Thus, if a change impacts a metamodel or a transformation, then the com-

Goal	Description
Goal 1 (G1)	Concern-specific Modularization
Goal 2 (G2)	Specifying multiple concerns at high-level of abstraction
Goal 3 (G3)	Enabling an oblivious mechanism to integrate new concern-specific requirements

Table 3.1: Research Goals

plexity of maintaining the consistency among all the artifacts will be increased [KKS07]. Therefore, the strategy used must minimize the changes to the existing MTC artifacts reducing the impact of adding a new concern. This is one of the most desired criterion in order to reduce the complexity of change. Therefore, we must identify the *impacted artifacts* (i.e., metamodels, models and transformations).

Related Goals: *G1: Concern-specific Modularization*, *G2: Specifying multiple concerns at high-level of abstraction*, and *G3: Enabling an oblivious mechanism to integrate new concern-specific requirements.*

The question to answer is: *Do the existing metamodels, models and transformation remain unchanged?*

Criterion 2 (C2): Use of high-level concern-specific concepts

The use of *Domain-Specific Modeling Languages* (DSMLs) increases the level of abstraction and gives domain experts suitable concepts for specifying an application close to the problem domain [TK05]. The use of *high-level concern-specific concepts* will allow the concern experts to specify the new concern requirements close to the concern domain. This means that MTC metamodel(s) should offer concern concepts that allow us to model the new requirements. These new concern concepts should be defined at a correct level of abstraction.

Related Goals: *G2: Specifying multiple concerns at high-level of abstraction.*

The question to answer is: *Are concern-specific concepts available in the metamodel(s) to specify the added concern?*

Criterion 3 (C3): Metamodel pollution

As presented in Section 1.1.1, adding concepts to a metamodel that clearly do not align with its existing concepts is detrimental to the metamodel understandability and maintainability. A key criterion to analyze a strategy for evolving an MTC is to avoid *metamodel pollution.*

Related Goals: *G1: Concern-specific Modularization*, and *G2: Specifying multiple concerns at high-level of abstraction.*

Criterion	Goals	Question
C1: Impacted Artifacts	G1, G2	G3 Do the existing metamodels, models and transformations remain unchanged?
C2: High-level concern concepts	G2	Are concern-specific concepts available in the metamodel(s) to specify the added concern?
C3: Metamodel Pollution	G1, G2	Are the metamodels free of alien concepts that do not belong to their domain?
C4: Monolithic Models	G1	Is a set of concern-specific models used to specify the whole application?
C5: Number of impacted elements	G3	Are the model elements impacted by the new concern specified at a high-level of abstraction?
C6: Identification complexity	G3	Is the new concern specified at the same level of abstraction as the elements it affects?
C7: Integration mechanism	G1, G3	Is it possible to use a common integration mechanism?

Table 3.2: Key criteria

The question to answer is: *Are the metamodels free of alien concepts that do not belong to their domain?*

Criterion 4 (C4): Monolithic model

Section 1.1.1 explains that adding specifications of new concerns to an existing model increases the complexity of the model and makes it more difficult to comprehend and maintain. It is undesirable to specify the complete application in a single *monolithic model*. The use of several concern-specific models follows the SoC principle reducing the complexity of specifying and evolving each model [Rea89, KLM+97]. Additionally, the existing models are unchanged and the new concerns are added externally.

Related Goals: *G1: Concern-specific Modularization.*

The question to answer is: *Is a set of concern-specific models used to specify the whole application?*

Criterion 5 (C5): Number of impacted model elements

At a high-level of abstraction, a large number of implementation details are excluded. In addition to the translation from problem domain concepts to solution domain concepts, the transformation chain adds implementation details in every transformation step. This results in a reduced set of elements in high-level models with respect to the low-level models. This implies that the *number of impacted elements* is lower in higher-levels than in lower-levels. Therefore, identifying the changed model elements will be a simpler task at the high-level than in the lowest level.

Related Goals: *G3: Enabling an oblivious mechanism to integrate new concern-specific requirements.*

The question to answer is: *Are the model elements impacted by the new concern specified at a high-level of abstraction?*

Criterion 6 (C6): Complexity of identifying the impacted model elements

When the new concern requirements are specified at a different level of abstraction than the existing elements that they affect, you need to have knowledge of the two abstraction levels involved (e.g., concern level and platform level). This will increase the *complexity of identifying the impacted model elements*. The ideal scenario would be to define the changes and identify the changed elements at the same level of abstraction.

Related Goals: *G3: Enabling an oblivious mechanism to integrate new concern-specific requirements.*

The question to answer is: *Is the new concern specified at the same level of abstraction as the elements it affects?*

Criterion 7 (C7): Common integration mechanism

Once the new requirements are specified they need to be integrated with the existing artifacts. Having in mind to reuse the evolution strategy with several existing MTCs and multiple added concerns, a *common integration mechanism* should be used. This mechanism should be metamodel independent in order to be reusable for adding different concerns to an existing MTC.

Related Goals: *G1: Concern-specific Modularization*, and *G3: Enabling an oblivious mechanism to integrate new concern-specific requirements.*

The question to answer is: *Is it possible to use a common integration mechanism?*

3.4 Evolution Strategies

Studying several works in other fields such as AOP, AOM, and multi-modeling, we found five strategies that use multiple concern specifications to describe an application. All have in common that the specifications should be integrated in order to obtain the required application. We have adapted these strategies in order to evolve an existing MTC. These strategies have both advantages and drawbacks. We will analyze the different strategies using the criteria presented before in order to identify the most appropriate one to evolve an MTC. It is important to consider that these strategies are used in other research fields in the context of describing an application using multiple concern specifications.

Figure 3.8 shows a schema of each strategy. In this figure it is possible to see where (i.e., high-level vs. low-level) the new concern is added, how it is related with the existing artifacts and where it is integrated with the existing MTC. The strategies are:

(a) *Extending the high-level metamodel* (presented in Figure 3.8(a)). In this strategy, new concepts are directly added to the existing metamodel and they are used to express the new requirements.

(b) *Composing high-level models* (presented in Figure 3.8(b)). In this strategy, a new concern model is added at a high-level of abstraction, and immediately composed with the existing high-level model.

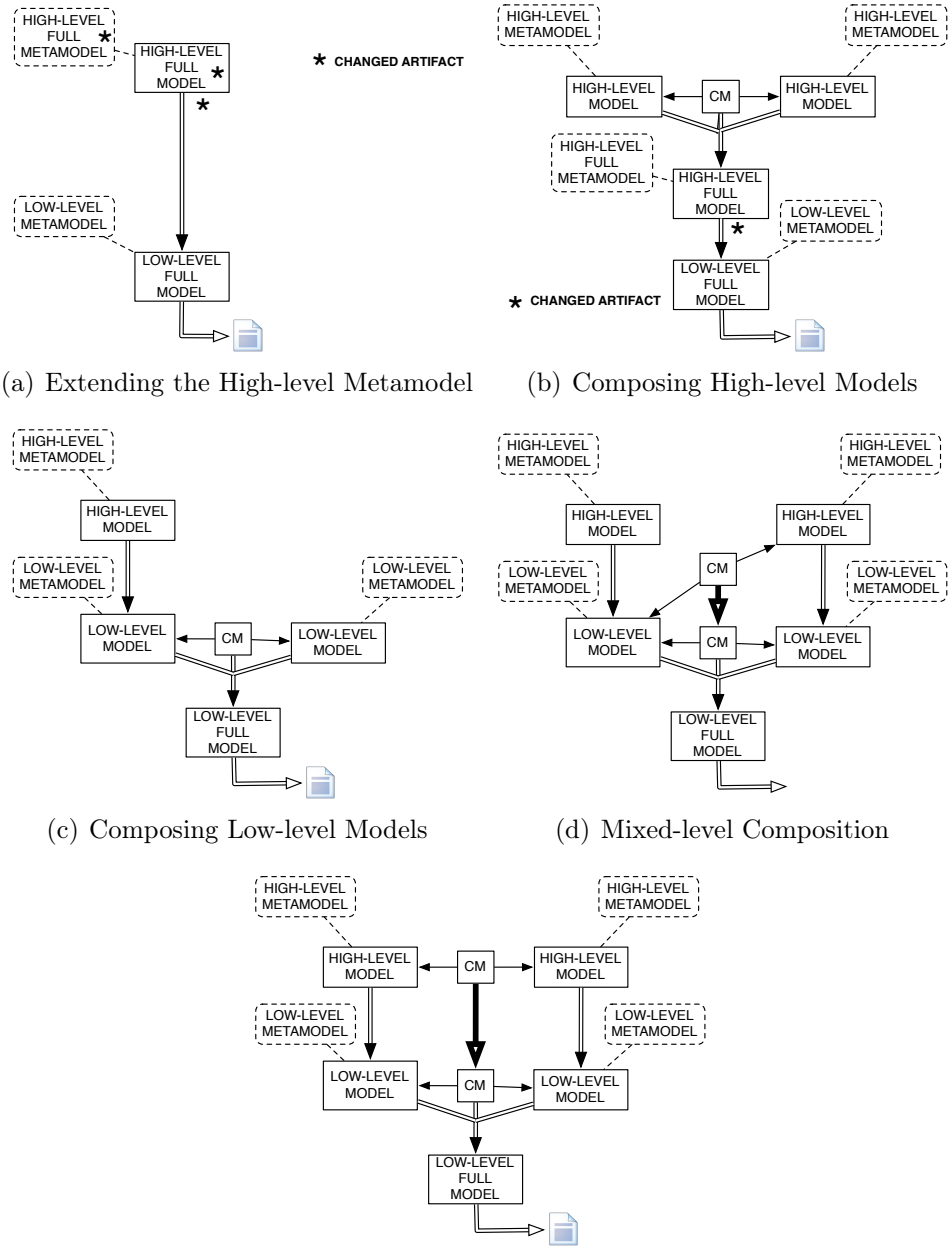
(c) *Composing low-level models* (presented in Figure 3.8(c)). In this strategy, a low-level concern model is added and composed with the existing low-level model.

(d) *Mixed-level composition* (presented in Figure 3.8(d)). In this strategy, a high-level concern model is added and it is related with the existing low-level model, after which the new concern model is transformed and composed.

(e) *Parallel model transformation chain* (presented in Figure 3.8(e)). In this strategy, a new concern-specific MTC is added and aligned with the existing one. The alignment is propagated through the whole MTCs and is used to guide the composition of the low-level models.

3.4.1 Extending the High-level Metamodel

The first strategy is to directly extend the high-level metamodel with the new concern concepts. For instance, to extend the Business metamodel with authorization concepts. Figure 3.9 shows the detailed strategy schema and the impacted artifacts.



(a) Extending the High-level Metamodel

(b) Composing High-level Models

(c) Composing Low-level Models

(d) Mixed-level Composition

(e) Parallel Model Transformation Chains

Figure 3.8: Evolution strategies

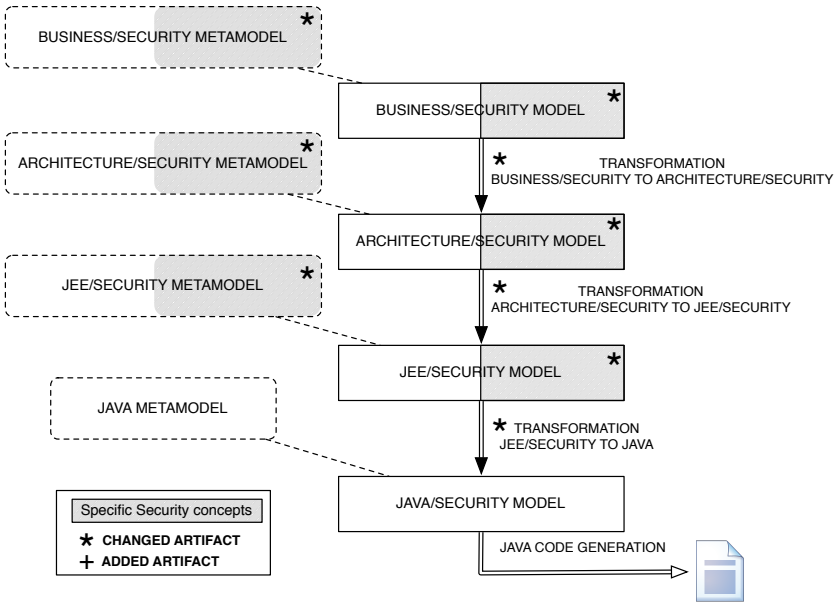


Figure 3.9: Extending the High-level Metamodel

In [BDL06], an approach called *Model-Driven Security* is presented. The goal of this research is to model an application and its security requirements using a high-level modeling language. This approach presents a general schema that is called *dialect* for constructing languages that combine system modeling languages with a security modeling language. Using this approach, an entire application can be specified together with access control requirements.

In order to extend the existing MTC using this approach, the first step is to extend the Business metamodel using the *dialect* mechanism and to add the concepts of the *SecureUML* metamodel [LBD02] to it. The SecureUML metamodel contains concepts that belong to the authorization concern. Applying this approach we obtain a single adapted metamodel that allows us to express the complete application and its security requirements in a single model. However, extending a metamodel with concepts from a different domain makes the metamodel polluted with alien concepts, increasing its complexity and reducing its comprehensibility, as presented in Section 1.1.1.

Once the extended metamodel (i.e. Business Metamodel and Security metamodel) is defined, it is possible to use the new concepts to specify the authorization policies of the application in the same model in which the business concepts are specified. As presented in Section 1.1.2, using a single monolithic model that expresses every involved concern in the application increases the complexity of evolving it.

Additionally, not only the Business metamodel should be extended, but the Architectural model and the JEE metamodel should be extended as well having in mind to include the security concepts at their specific level of abstraction

(i.e., architectural level, Java application server level). The Java metamodel does not need to be extended because it offers the required concepts to express authorization (i.e., Java annotations).

It is obvious that if the Business metamodel, the Architectural metamodel and the JEE metamodel change, the Business to Architecture ($T_{bus2arch}$) and the Architecture to JEE ($T_{arch2jee}$) transformations must be modified as well. Finally, the JEE to Java transformation ($T_{jee2java}$) must be modified due to the changes to the JEE Metamodel and to be able to generate security annotations with the required authorization policies in the Java model. The Code Generation (G_{java}) does not require to be extended because the fact that the Java metamodel is unchanged and that Java annotations belong to it.

In conclusion, this strategy is only suitable also for evolving a small MTC (i.e., small set of metamodel elements, simple transformations). However, it is not appropriate for adding a new concern into large MTC. Because this strategy requires the modification of almost every metamodel and transformation to be able to manage the new security concepts we consider this strategy inappropriate to evolve an existing MTC. The use of this approach is more appropriate for a new MTC built with security in mind. Table 3.3 shows a summary of the criteria analysis. Below, we briefly discuss the criteria for this strategy.

Criterion 1 (C1): Impacted Artifacts

This strategy is highly invasive. It requires the modification of almost every MTC existing artifact in order to introduce the new concern. The modified artifacts are: Business Metamodel ($MM_{business}$), Architecture Metamodel ($MM_{architecture}$), JEE Metamodel (MM_{jee}), Business model ($M_{business}$), Business to Architecture Transformation ($T_{bus2arch}$), Architecture to JEE Transformation ($T_{arch2jee}$), and JEE to Java Transformation ($T_{jee2java}$). Therefore, this criterion is **negative**.

Criterion 2 (C2): Use of high-level concern-specific concepts

This strategy offers high-level concepts to specify the authentication policies. This allows security experts to specify their requirements using suitable concepts. Therefore, this criterion is **positive**.

Criterion 3 (C3): Metamodel pollution

The Business metamodel is polluted with Security concepts. This reduces the comprehensibility of the metamodel and increases the complexity of evolving it as well as of specifying models that conform to it. Therefore, this criterion is **negative**.

Criterion 4 (C4): Monolithic Model

The Security authentication policies are specified in the same model as the Business specification. This increases the complexity of specifying the application and its security. Additionally, having a monolithic model makes the evolution of the required application a difficult task. Therefore, this criterion is **negative**.

Criterion 5 (C5): Number of impacted model elements

The model elements affected by the new requirement are identified at the high-level model, thus the number of changed elements are less than in the lower level models. Therefore, the tasks that the *application modeler* should perform in order to identify the elements to be secured is simpler than in a lower level. Therefore, this criterion is **positive**.

Criterion 6 (C6): Complexity of identifying the impacted model elements

The new requirements and the changed elements are at the same level of abstraction. This is to change the high-level Business model adding the authorization policies to its elements. Therefore, this criterion is **positive**.

Criterion 7 (C7): Common integration mechanism

The changes are manually added into the existing business model, metamodel and transformations. If an additional concern is added, then it will be necessary to manually add in the changes again. Therefore, this criterion is **negative**.

3.4.2 Composing High-level Models

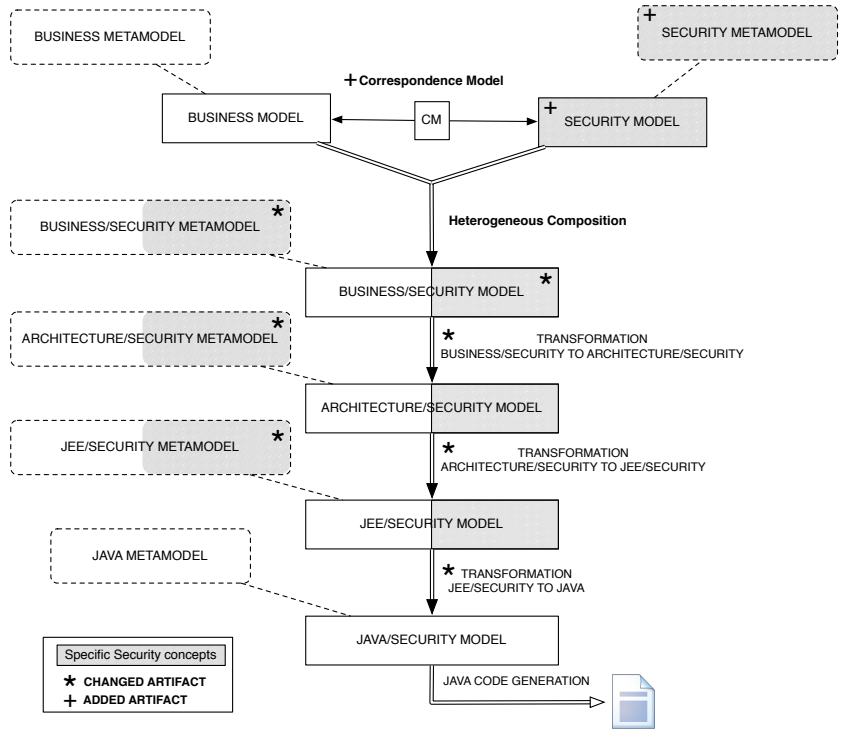
Another possible strategy is to add a new independent high-level model where the new security requirements are specified. This new model conforms to a security metamodel and it will allow the concern experts to specify the new requirements using authorization concepts. Using this strategy, it is possible to maintain the Business Metamodel clean and the Business model oblivious of the added concern models. Having separate models reduces the complexity of specifying and evolving the modeled applications. For instance, in the presented case study, we have a Business model and a Security model. As a Business model it is possible to use one of the existing business models (e.g., the `Risk` business model), and protect it without any direct modification. The Security model can be defined by a security expert using the SecureUML metamodel independently. In the Security model the authorization policies required to protect the application can be specified. There are two possible options to

bring the two high-level models to the platform specific level. The first one is to directly compose the models into a merged high-level model and then transform that model into a low-level model. Figure 3.10(a) shows the detailed schema of this strategy when the models are composed into a high-level merged model. The second option is to transform and compose the two high-level models into a low-level model. Figure 3.10(b) shows the detailed schema of this strategy when the models are composed and translated into a low-level platform model at the same time.

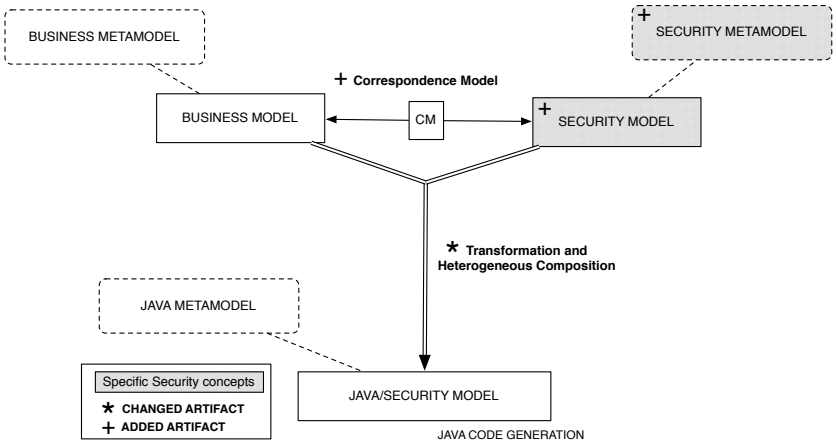
Once both models are defined, it is necessary to align them. This can be done by using correspondence relationships between the models. These relationships allow the *application modeler* to identify the corresponding elements in each model. The modeler must identify the corresponding elements and obtain the correspondence model.

Once the Business model and the Security model are aligned using correspondence relationships, it is necessary to integrate the models. This is done by performing a composition. However, at a high-level of abstraction, both models conform to different metamodels and a heterogeneous composition is required. This means that one has to define how each pair of business and security concepts should be composed. For instance, a *BusinessEntity* with a secured *Resource*, an *Attribute* with a secured *Resource*, etc. A heterogeneous composition requires the definition of a composition semantics for every pair of involved concerns and the implementation of a composition mechanism that increases the complexity of evolving an MTC.

After the two models are composed, it is possible to obtain a model that conforms to a metamodel containing concepts of the business domain and the security domain similar to the extended metamodel used by the previous strategy. This means that a high-level metamodel that allows to express every involved concern must be used. This kind of composition is used in [CDR08], in which a Web application is specified using three high-level models: a Data model where the data structures and their relationships are specified, a Navigation Model where the navigation paths are specified, and a Composition model where the Web pages' structure is specified. These three models are aligned using correspondences between them and eventually composed into a single model that conforms to a combined metamodel containing concepts of the three domains (i.e., data, navigation and composition). Finally, the composed high-level model should be transformed into a low-level platform specific model in order to obtain the required application. The goal of the research presented in [CDR08] is to reduce the complexity of specifying and evolving web applications, but it does not consider the evolution of the used MTC. In our case study the use of this kind of composition will bring us back to the previous strategy, starting from a single model where the Business and Security are specified at the same time. Therefore, we will need to extend the MTC metamodels and transformations in the same way as the previous strategy.



(a) High-level composition metamodel



(b) High-level composition

Figure 3.10: Composing High-level Models

An additional situation can be presented by composing both models and producing a low-level platform model. This avoids having a combined high-level metamodel with every involved concern in it. However, this makes us develop a completely new set of transformations that should compose and translate the models to the platform level at the same time. This ends up making the MTC even more complex and difficult to maintain.

In conclusion, this strategy requires us to modify almost every MTC artifact, similar to the previous strategy. Additionally, this strategy requires us to perform a heterogeneous composition, increasing the complexity of the MTC. Therefore, this strategy is not appropriate for adding a new concern to an existing MTC. The use of this approach is more appropriate when the concerns involved in the application specification are fixed and a merged metamodel is available for those concerns. The evaluation of our criteria for this strategy is presented in Table 3.3. Below, we briefly discuss the criteria for this strategy.

Criterion 1 (C1): Impacted Artifacts

This approach is highly invasive. Any of the possible options (Figure 3.10) requires us to modify almost every existing MTC artifact in order to introduce the new concern. The modified artifacts are: Business Metamodel ($MM_{business}$), Architecture Metamodel ($MM_{architecture}$), JEE Metamodel (MM_{jee}), Business model ($M_{business}$), Business to Architecture Transformation ($T_{bus2arch}$), Architecture to JEE Transformation ($T_{arch2jee}$), and JEE to Java Transformation ($T_{jee2java}$). Therefore, this criterion is **negative**.

Criterion 2 (C2): Use of high-level concern-specific concepts

This approach offers high-level concepts to specify the authentication policies in an independent metamodel. This allows security experts to specify their requirements using suitable concepts. Therefore, this criterion is **positive**.

Criterion 3 (C3): Metamodel pollution

The Business metamodel is free of security concepts. This helps the *application modeler* to understand each metamodel, and to define models that conform to them. Therefore, this criterion is **positive**.

Criterion 4 (C4): Monolithic Model

The security authentication policies are specified independently of the Business specification. This reduces the complexity of specifying the application and its security. Therefore, this criterion is **positive**.

Criterion 5 (C5): Number of impacted model elements

The elements affected by the new requirements are identified at the high-level model, thus the number of changed elements are less than in the lower-level models. Therefore, the tasks that the modeler should perform identifying the elements to secure is simpler than in the lowest level. Therefore, this criterion is **positive**.

Criterion 6 (C6): Complexity of identifying the impacted model elements

The new requirement specification and the changed elements are at the same level of abstraction. A high-level security model is defined and related to the existing business model, identifying the changed elements. Therefore, this criterion is **positive**.

Criterion 7 (C7): Common integration mechanism

The composition of both specifications (i.e, business and security) is performed by special-purpose transformation that is developed for this specific pair of metamodels. It cannot be reused for a different concern. Therefore, this criterion is **negative**.

3.4.3 Composing Low-level Models

In the previous approaches, if a new concern cannot be expressed using the existing metamodels, it is impossible to add a new concern to the MTC without changing it. Having in mind to avoid changing the existing metamodels, models and transformations, it is necessary to find the best point to introduce the changes, minimizing the modifications to the MTC.

In order to avoid triggering the *ripple effect*, it is clear that the least intrusive point is after the last model-to-model transformation. At this point it is possible to keep all the metamodels, models and transformation unchanged. This strategy is similar to traditional AOP where the main application and the concern are specified using the same language.

When the lowest-level model (i.e., Java model) is produced by the existing MTC, it is possible to specify the new concern requirements in a model that conforms to the same platform metamodel (e.g., the Java metamodel) as the existing model. The Java metamodel allows the *application modelers* to express all the application concerns without any modification. For instance, the security mechanism used are Java annotations that are concepts that belong to the Java metamodel.

Once the security requirements are specified using the Java metamodel, it is necessary to identify the place in the existing Java model where they need to

be introduced. Similar to the previous strategy, we can use a correspondence model to identify where we need to change the existing application model. In the case study this means we must identify which methods will be protected by Java security annotations. Finally, both models can be composed in order to obtain a secure application model. Due to the fact that both models conform to the same metamodel, it is possible to perform a homogeneous composition. This means that it is possible to use the same composition mechanism for the concerns that can be expressed using the Java metamodel. Using the same composition mechanism allows the *MTC developers* to reuse it with several concerns. Figure 3.11 shows how the low-level security model is added after the lowest model was generated. This model contains all the platform-specific implementation details for the application and it can be composed with the low-level security model.

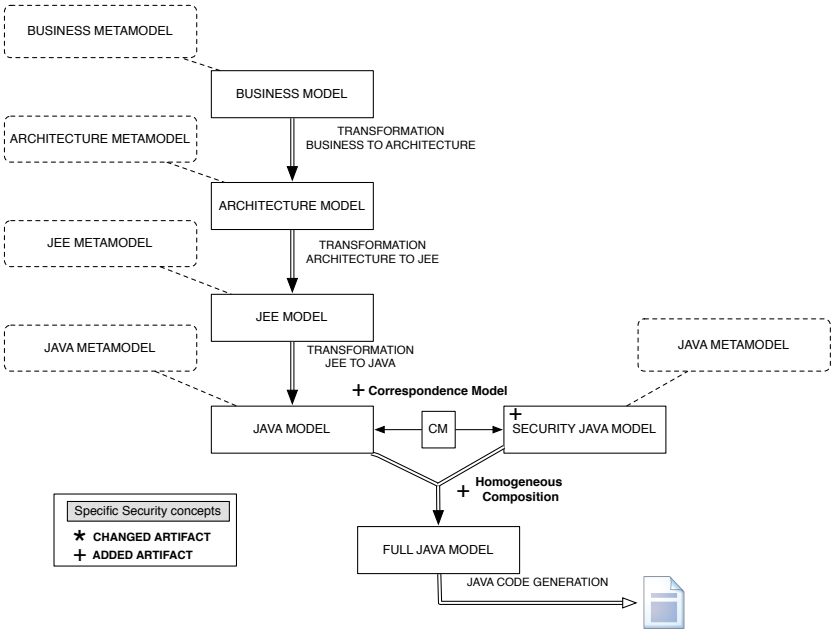


Figure 3.11: Composing Low-level Models

Unfortunately, at a high-level of abstraction, a large number of implementation details are excluded, and the transformation chain adds implementation details in every transformation step. This results in a higher set of elements in the low-level models with respect to the high-level models. Therefore, specifying a security model using Java concepts is an overwhelming task where you actually lose most of the benefits of using a model-driven approach (i.e. you are programming not modeling).

In conclusion, this strategy is not appropriate for adding a new concern to an existing MTC. This strategy requires knowledge about the technology and how the application is built in order to specify the new concern. Moreover,

defining the correspondences at this level is an overwhelming task because of the higher number of elements in the low-level models with respect to higher level models. The evaluation of our criteria for this strategy is presented in Table 3.3. Below, we briefly discuss the criteria for this strategy.

Criterion 1 (C1): Impacted Artifacts

This approach is slightly invasive. It does not modify any existing artifact by adding an independent security model at the lowest level of the MTC. At the lowest level, the existing MTC is completely reused and a common composition mechanism can be used to perform the integration of the security specification with the application model. Therefore, this criterion is **positive**.

Criterion 2 (C2): Use of high-level concern-specific concepts

This approach uses low-level concepts (e.g., Java concepts) to specify the authentication policies. This increases the complexity of specifying the security. Therefore, this criterion is **negative**.

Criterion 3 (C3): Metamodel pollution

The Business metamodel is not polluted with security concepts. This helps in using the business metamodel, and in evolving the models that conform to it. Therefore, this criterion is **positive**.

Criterion 4 (C4): Monolithic Model

The security authentication policies are specified independently of the existing models. This reduces the complexity of specifying the application and its security. Therefore, this criterion is **positive**.

Criterion 5 (C5): Number of impacted model elements

The elements affected by the new requirement are identified at the low-level model, thus the number of changed elements are greater than in the higher level models. Furthermore, the tasks that the modeler should perform in order to identify the elements that need to be secured are more complex compared to if it had been done at the highest level. Therefore, this criterion is **negative**.

Criterion 6 (C6): Complexity of identifying the impacted model elements

The changes and the changed elements are at the same level of abstraction. This means that the existing low-level models are changed by adding the au-

thorization policies that are specified at the same level of abstraction (e.g., Java annotations are added to Java methods). Therefore, this criterion is **positive**.

Criterion 7 (C7): Common integration mechanism

At the lowest level, it is possible to use a common integration mechanism in order to perform a homogeneous composition. Therefore, this criterion is **positive**.

3.4.4 Mixed-level Composition

At its main advantage, the *High-level composition* strategy presented in Section 3.4.2 offers the use of an independent high-level metamodel that allows the domain experts to express the new concern requirements in a suitable way. The main advantage of the *Low-level composition* strategy is to reduce the impact on the existing artifacts by introducing the changes after the last model-to-model transformation. These two characteristics are essential in reducing the complexity of evolving an MTC and in offering appropriate concepts to the concern experts to be able to express their requirements. The *Mixed-level composition* strategy has these two advantages, as it offers a high-level language to the concern experts as well as reduces the impact on the existing artifacts.

The idea of the *Mixed-level composition* is to add a new high-level model in which the new concern requirements are specified. This strategy is used in [FTD08]. This work presents an approach that modularizes transactions as aspects and specifies them with a high-level transaction language. The high-level transaction aspect is related to low-level application models using *pointcut* expressions. Next, the transaction aspect is refined and a low-level aspect is generated. Finally, the low-level aspect is weaved with a low-level application.

If we use these ideas to add security to the MTC presented in Section 3.2, a Security metamodel is required, and the security experts should define a new Security model. Figure 3.12 shows the *Mixed-level composition* of the original MTC with a high-level Security model. In this model, all of the elements of the security specification must be expressed using high-level security concepts. Using this strategy, it is necessary to define correspondences between the high-level Security model and the low-level Java model. For instance, a high-level security permission such as *Read* must be related to all the getter *Methods* in the Java model. The next action is transforming the high-level Security model into a low-level Java Model. Additionally, it is necessary to derive correspondences between the Java model and the Java security model using the correspondences defined between the high-level Security model and the Java model. Finally, a homogeneous composition is performed between both

low-level Java models, producing a full Java model. This low-level Java model contains all of the specific platform concepts for the application and its access control mechanisms (i.e., Java annotations).

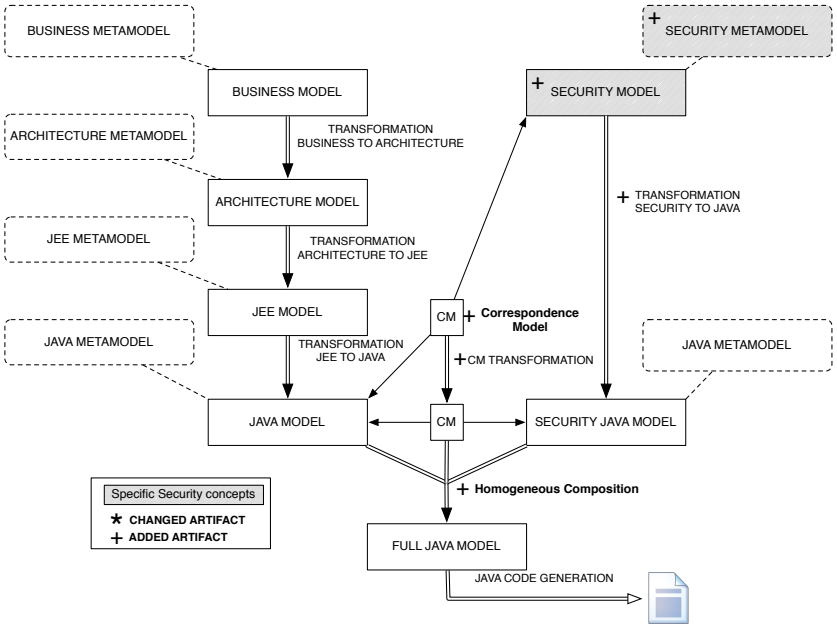


Figure 3.12: Mixed-level Composition

However, this strategy has a drawback due to the definition of the correspondence model between two different levels of abstraction. This increases the complexity because the modeler must define correspondence between elements that belong to the high-level Security model and elements that belong to the low-level Java model. Therefore, the modeler must have knowledge of the concern concepts as well as the technological platform, which goes against the MDE goals.

In summary, this strategy offers high-level concepts in order to specify the security and minimize the changes in the existing MTC. However, it requires a high effort to be able to specify the correspondence relationships between the high-level Security model and the low-level Java model. Consequently, this strategy is not suitable to evolve an existing MTC. The evaluation of our criteria for this strategy is presented in Table 3.3. Below, we briefly discuss the criteria for this strategy.

Criterion 1 (C1): Impacted Artifacts

This approach is non-invasive. It does not modify any existing artifact by specifying an independent high-level security model. This model is transformed into a low-level model and composed with the existing low-level model. Using this

strategy, the existing MTC is completely reused and a common composition mechanism can be used to perform the integration of the security specification with the application model. Therefore, this criterion is **positive**.

Criterion 2 (C2): Use of high-level concern-specific concepts

This approach uses a high-level metamodel to express the new concern. This reduces the complexity of specifying the authentication policies. Therefore, this criterion is **positive**.

Criterion 3 (C3): Metamodel pollution

The Business metamodel is free of security concepts. This helps in understanding each metamodel, and in defining models that conform to them. Therefore, this criterion is **positive**.

Criterion 4 (C4): Monolithic Model

The security authentication policies are specified independently of the Business specification. This reduces the complexity of specifying the application and its security. Therefore, this criterion is **positive**.

Criterion 5 (C5): Number of impacted model elements

The elements affected by the new requirement are identified at the low-level model, thus the number of changed elements are greater than in the higher level models. Furthermore, the tasks that the modeler should perform in order to identify the elements that need to be secured are more complex compared to if it would be done at the highest level. Therefore, this criterion is **negative**.

Criterion 6 (C6): Complexity of identifying the impacted model elements

The requirements and the changed elements are at different levels of abstraction. This increases the complexity of evolving the MTC. Therefore, this criterion is **negative**.

Criterion 7 (C7): Common integration mechanism

At the lowest level, it is possible to use a common integration mechanism to perform a homogeneous composition. Therefore, this criterion is **positive**.

3.4.5 Parallel Model Transformation Chains

The last strategy that we analyze is the *Parallel Model Transformation Chain*. In this strategy, a new MTC is added and then aligned with the existing MTC. Both MTCs transform a pair of high-level models into two complementary low-level models. These low-level models are composed by taking advantage of the fact that at the lowest level: 1) the existing MTC artifacts are unmodified, and 2) it is possible to use a common low-level metamodel for both models.

Figure 3.13 shows this strategy. In this strategy a high-level security model is defined independently of the existing business high-level model. The new security model conforms to a security metamodel that has the required authorization concepts. The *application modeler* must define correspondence relationships between the existing high-level business model and the new security model by taking advantage of the fact that these two models are at the same level of abstraction. Additionally, as previously explained, high-level models exclude implementation details. This results in a smaller number of elements in high-level models with respect to the low-level models. This means that the modeler needs to identify a smaller number of correspondences at a high-level.

Once both models and the correspondences between them are specified, the existing model is transformed by the existing MTC into a low-level model and the new security model is transformed by the new specific security MTC into a Java model. These low-level models need to be composed to generate the final application and the *application modeler* needs to identify the elements to compose by defining correspondence relationships between the low-level models. These correspondence relationships between the low-level models are automatically derived by a transformation using the high-level correspondence relationships. Using the low-level correspondence relationships, a homogeneous composition is performed and a full Java model is obtained with the application specification and the new concern.

This strategy is used in the work presented in [CD06]. This work uses this strategy to specify business rules as a concern. The business rules are modeled using a high-level modeling language and then related to the application model using a high-level connection language. This connection language abstracts the different AOP patterns of how the business rules are connected with the application code. Next, both models are refined to low-level models. The connections are also refined and aspect code is generated from it.

In summary, this strategy offers several advantages to evolve an MTC. First, the existing artifacts are unchanged because the new concern is introduced after the last model-to-model transformation. Second, the new concern is modeled independently by using security specific concepts that belong to an independent security metamodel. This means that several smaller models are used to specify the required application and that the metamodels are not polluted. Third, the composition is postponed to the lowest level of abstrac-

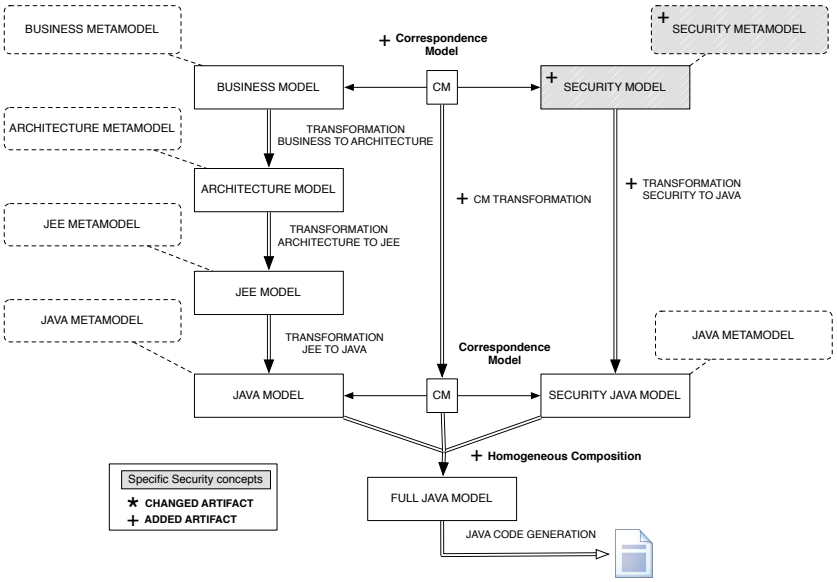


Figure 3.13: Parallel Model Transformation Chain

tion where both models conform to the same metamodel. This means that a homogeneous composition can be performed. However, a generalization of this strategy is required in order to use it for adding new concerns to existing MTCs. The evaluation of our criteria for this strategy is presented in Table 3.3. Below, we briefly discuss the criteria for this strategy.

Criterion 1 (C1): Impacted Artifacts

This approach is non-invasive. It does not modify any existing artifact by specifying an independent high-level security model. Rather, this model is transformed into a low-level model and composed with the existing low-level model. Using this strategy, the existing MTC is completely reused and a composition mechanism can be used to perform the integration of the security specification with the application model. Therefore, this criterion is **positive**.

Criterion 2 (C2): Use of high-level concern-specific concepts

This approach uses a high-level metamodel to express the new concern. This reduces the complexity of specifying the authentication policies. Therefore, this criterion is **positive**.

Criterion 3 (C3): Metamodel pollution

The Business metamodel is free of security concepts. This helps in understanding each metamodel, and in defining models that conform to them. Therefore,

this criterion is **positive**.

Criterion 4 (C4): Monolithic Model

The security authentication policies are specified independently of the Business specification. This reduces the complexity of specifying the application and its security. Therefore, this criterion is **positive**.

Criterion 5 (C5): Number of impacted model elements

The elements affected by the new requirement are identified at the high-level model, thus the number of changed elements are fewer than in the lower level models. Therefore, the tasks that the modeler should perform in order to identify the elements that need to be secured are less complex than at the lowest level. Therefore, this criterion is **positive**.

Criterion 6 (C6): Complexity of identifying the impacted model elements

The new requirements and the changed elements are at the same level of abstraction. This reduces the complexity of evolving the MTC. Therefore, this criterion is **positive**.

Criterion 7 (C7): Common integration mechanism

At the lowest level, it is possible to use a common integration mechanism to perform a homogeneous composition. Therefore, this criterion is **positive**.

3.5 Summary

Table 3.3 presents a summary with the key criteria analysis for each strategy. This table indicates that the worst strategy for our problem context is *Extending the High-level Metamodel* because this strategy fails in several key criterion. Additionally, this table indicates that the best strategy is *Parallel Model Transformation Chain*.

This strategy succeeds in every key criterion and it can be generalized to fulfill our research goals presented in Section 1.2. Although in the work of Cibran [CD06] this strategy is used to specify and weave a concern independently of the main application, we need to translate its ideas to the MDE context and specifically to the evolution of an existing MTC. Additionally, several elements must be generalized in order obtain a non-invasive and reusable approach to evolve existing MTCs.

	Metamodel Extension	High-level Comp.	Low-level Comp.	Mixed-level Comp.	Parallel Model Transf.
C1: Impacted Artifacts	−	−	+	+	+
C2: High-level concern concepts	+	+	−	+	+
C3: Polluted Metamodel	−	+	+	+	+
C4: Monolithic Model	−	+	+	+	+
C5: Number of impacted elements	+	+	−	−	+
C6: Identification complexity	+	+	+	−	+
C7: Integration mechanism	−	−	+	+	+

Table 3.3: Comparative analysis

The focus of our research is to generalize the mechanism to automatically obtain a low-level CM and to compose the low-level homogeneous models. The generalization of the mechanism to automatically obtain a low-level CM is presented in Chapter 4. In Chapter 5 we present the use of CMs to integrate different concerns with the existing MTC at lowest level of abstraction reducing the impact on the existing artifacts. In Chapter 6 we present the tool support that we provide to help *MTC developers* and *application modelers* to use our approach. Finally, in Chapter 7 we present the validation of our work with the help of a full-fledged case study.

Chapter 4
Correspondence Relationships Derivation

4.1 Introduction

In the previous chapter, we presented an analysis of possible strategies that can be used to evolve an existing MTC. These strategies were extracted from several works that specify concerns independently. The independent concerns are automatically integrated with the main application. From the analysis we found that the strategy that scored best in our criteria (presented in Section 3.3.1) is: *Parallel Model Transformation Chain.*

In this chapter we present our approach to evolve a MTC. This approach is based on the generalization of the *Parallel Model Transformation Chains* strategy. The goal of this generalization is to use this strategy to evolve existing MTCs with new concerns. The generalization of this strategy is based on the use of an automatic correspondence relationships derivation mechanism. This mechanism allows to keep two independent MTCs (i.e., the existing MTC and the new concern specific MTC) aligned through their whole set of transformation steps.

The automatic correspondence relationship derivation mechanism uses the correspondence relationships between high-level models to derive a new set of relationships between the generated low-level models. In order to produce the low-level correspondences, the mechanism uses *tracing models*(TMs) to identify elements in the low-level models that are generated from a couple of high-level elements related by a correspondence relationship. Additionally, the new set of correspondence relationships between the generated elements are constrained by a *Correspondence Derivation Model* (CDM). The CDM contains constraints that allow or restrict the generation of relationships between a couple of low-level elements. The reason of this is to only relate valid corresponding[1] elements in the low-level models and to avoid *false positives.*

[1]Elements that must be composed at the low-level of abstraction in order to obtain the full application.

In this chapter we use two examples to illustrate our proposal. The first example is presented in Section 4.2 and it focuses on showing our general strategy to evolve an existing MTC in a simple and abstract way. The second example is presented in Section 4.3 and it revisits the presented strategy but uses a concrete real-life example. This example shows in detail our ideas and explains the implementation details. Next, in Section 4.4 the requirements for the derivation mechanism are presented. In Section 4.5 we explain in detail how we generate and use tracing models. Next, in Section 4.6 we present how we constrain the derived correspondence relationships. Finally, in Section 4.7 we explain how we extend the scope of the Correspondence Derivation Mechanism.

4.2 Approach overview

The overall approach transforms two high-level models that conform to different metamodels into low-level models that conform to the same existing metamodel. To illustrate this, we use a very simple example where we use geometrical shapes as metaphors for metaclasses. This allows us to represent different metamodels and to leave out the details of the models and what they represent. We will retake the strategy with model-level details in Section 4.3.

In the example we want to model an application using two concerns. The first concern is the *Elliptical* domain and the second concern is the *Circular* domain. At the top part of the Figure 4.1 the two domains are depicted. The *Elliptical* and the *Circular* domains are at the problem level. Our application is implemented in the *Polygonal* platform. At the bottom part of the Figure 4.1 the *Polygonal* domain is depicted at solution level. One the one hand, the Elliptical expert specified a set of requirements using the concepts of the *Ellipse* metamodel. On the other hand, the Circular expert specified other requirements using concepts from the *Circle* metamodel. These two sets of requirements can be translated in terms of *Polygonal* domain concepts using MTCs. The *Polygonal* concepts are specified in the low-level *Polygon* metamodel. At this level we have several "platform" concepts, such as: *Square*, *Triangle*, *Diamond*, and *Pentagon*.

Figure 4.2 shows an MTC that transforms *Ellipse* models into *Polygon* models, and an MTC that transforms *Circle* models into *Polygon* models. The *Ellipse* and the *Circle* models represent two concerns of an application that need to be integrated to produce the full application. Both models are related by a correspondence model. This high-level model needs to be propagated through the complete transformation chains. The question is how we can obtain correspondence relationships between the low level models (i.e., the *Polygon* models). The main challenge is to define a mechanism that will automatically derive the new correspondence relationships, keeping in mind that the MTC increments the complexity of the models by adding elements at each

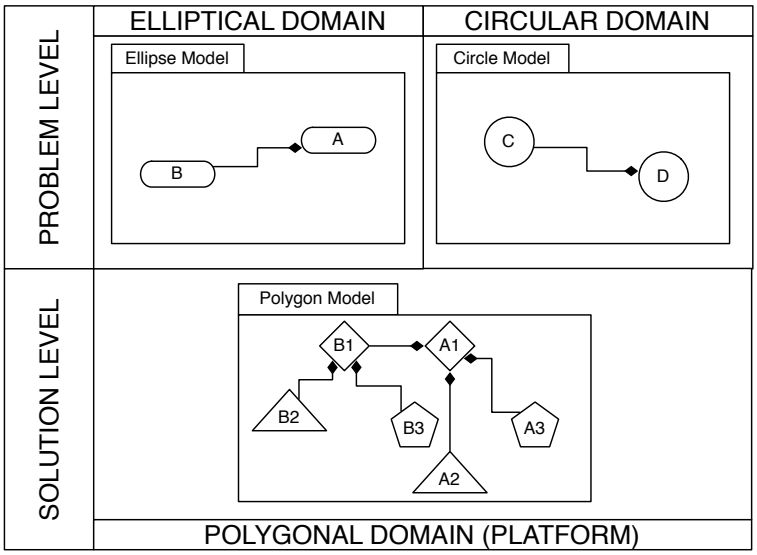


Figure 4.1: Geometrical Example

step.

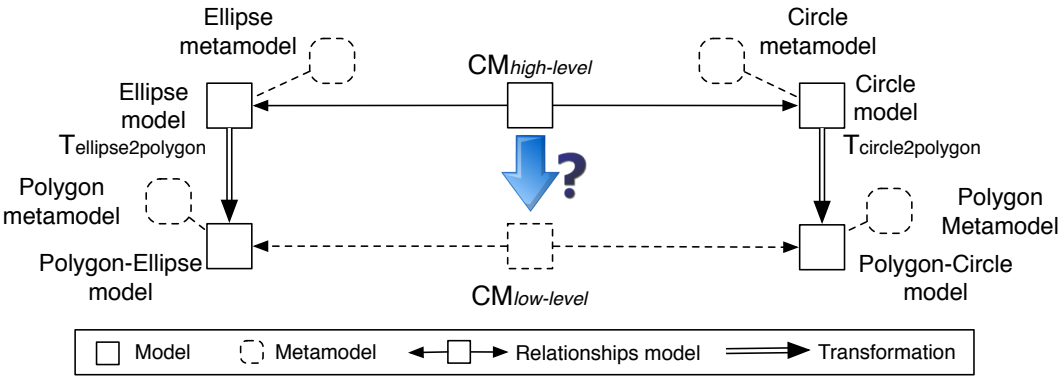


Figure 4.2: How to generate the low-level CM

4.2.1 High-level correspondences

At a high-level of abstraction, we have the `Ellipse` and the `Circle` models. These two models have some overlapping concepts that are aligned using a correspondence model (i.e., elements that partially represent the same "thing" in each domain). With the correspondence model we identify these corresponding concepts. Figure 4.3 shows the `Ellipse` and `Circle` high-level models. As we stated before, the model `Ellipse` and the model `Circle` conform to different metamodels. The model `Ellipse` has the following *Ellipse* concepts: `A` and `B`, and the model `Circle` has the following *Circle* concepts: `C` and `D`.

Additionally, the elements `A` and `C` represent the same element in different domains. Due to this `A` and `C` are related by a correspondence relationship. We want to integrate both models, but "intuitively" it is not possible to compose `Ellipse` elements with `Circle` elements because they belong to different metamodels. Therefore, a complex heterogeneous composition would be required. It is crucial to remember that what the models represent is not important in this example; what is important, however, are the corresponding elements and the metaclasses to which these conform to.

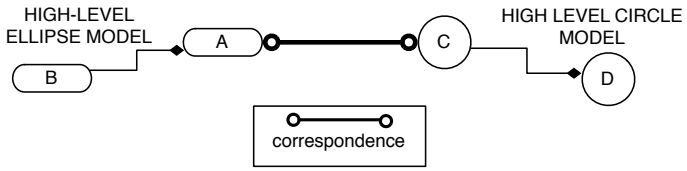


Figure 4.3: High-level correspondences

In order to avoid a heterogeneous composition, we transform both models into low-level models that conform to the same metamodel (i.e., *Polygon* metamodel). If both models conform to the same metamodel we can perform a homogeneous composition.

4.2.2 Tracing back to the sources

Once the MTCs are executed, two low-level models that conform to the *Polygon* metamodel are generated. We need to identify the corresponding elements in these low-level models based on their sources. Specifically we need to know if an element was generated from a high-level element that has a correspondence relationship. In order to verify this we need to *trace back* the elements of the low-level models and to check if they are generated from pairs of related elements at the higher-level. For instance, Figure 4.4 shows the two MTCs (*Tellipse2polygon* and *Tcircle2polygon*). These two MTCs transform the high-level models into low-level models that conform to the *Polygon* metamodel. On the one hand, the first MTC (*Tellipse2polygon*) transforms the elements `A` and `B` into the elements `A1`, `A2`, `A3`, `B1`, `B2` and `B3`. On the other hand, the second MTC (*Tcircle2polygon*) transforms the elements `C` and `D` into the elements `C1`, `C2`, `C3`, `D1`, `D2` and `D3`. Additionally, the high-level elements `A` and `C` are related by a correspondence relationship.

Keeping in mind the identification of the corresponding elements in the low-level models, we need a mechanism that can identify and verify that the source elements were related by a high-level correspondence relationship. For instance, in Figure 4.4 the elements `A1`, `A2`, `A3`, and the elements `C1`, `C2`, `C3` are generated from the elements `A` and `C`. With a *Tracing Model* (*TM*) [ARNRSG06] we are able to determine the elements in both low-level models

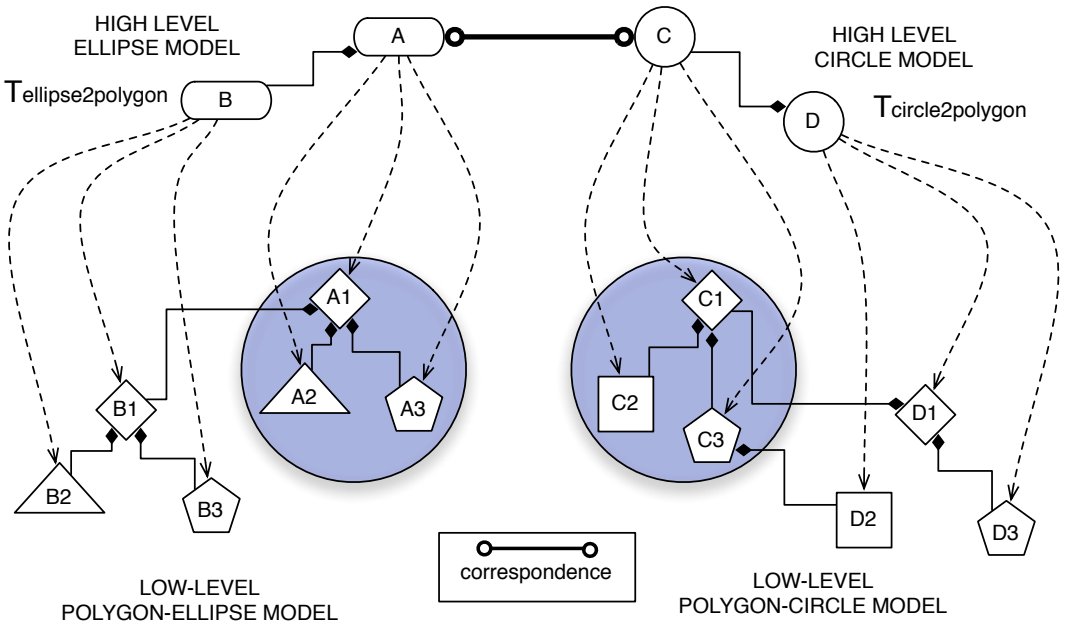


Figure 4.4: Traces

that were generated from a couple of related elements at the higher-level. How we use Tracing Models to trace back the elements generated by an MTC is presented in Section 4.5. Once again, the focus of this example is to identify which elements have corresponding relationships in the high-level models, into which elements these elements are transformed, and which metaclasses they conform to (i.e., which shape they have).

4.2.3 Constraining the relationships

Once we establish which elements in the low-level models are generated from a pair of corresponding elements in the high-level models, we must relate them to each other by identifying the *correct match* for each one. Figure 4.5 shows part of the low-level models `Polygon-Ellipse` and `Polygon-Circle`. The presented elements (in the colored circles) are generated from high-level corresponding elements that need to be correctly matched. Assume that only elements that conform to the same metaclass (i.e., have the same shape) can be related. Although in this case it is possible to define 9 possible relationships (e.g., `A1-C1`, `A1-C2`, ...) between the generated elements (in the colored circles), only the correspondence relationships between elements with the same shape are correct. In this example the correct match is: the *Diamond* `A1` with the *Diamond* `C1` and the *Pentagon* `A3` with the *Pentagon* `C3`. The element `A2` and the element `C2` cannot be related because they have different shapes. Although the element `B1` and the element `D1` have the same shape, they cannot

be related because they are generated from non-related high-level elements.

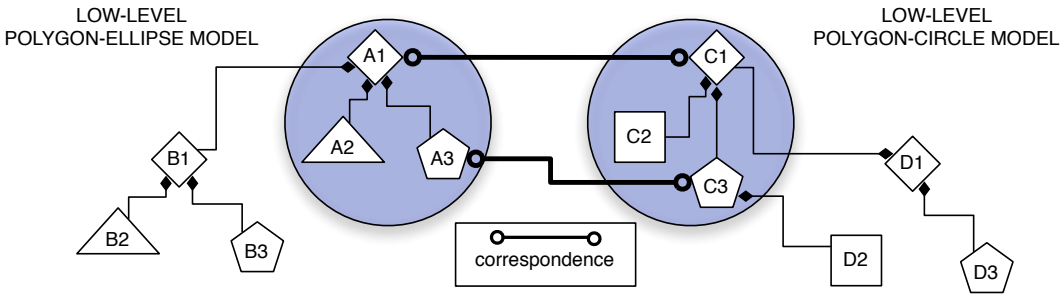


Figure 4.5: Corresponding elements

Therefore, we need a language that helps us to constrain the possible correspondence relationships that can be defined between the low-level models. We call this mechanism a *Correspondence Derivation model* (CDM) and it is presented in Section 4.6.1.

4.2.4 Correspondence relationships resolution

When the correspondence model is generated between the low-level models, a *resolution* of the correspondence relationships is required to obtain a full model. The term *resolution* in our work is defined as the interpretation of the correspondence relationships and processing them in order to produce the application. Suppose that in the presented example the resolution of a correspondence relationship is to *compose* the element in the left side of the correspondence relationship with the element in the right. Figure 4.6 shows the composition of the two low-level models. In this example, the composition of two corresponding elements (e.g., `A1` and `C1`) will produce in the full model only the element in the left side (e.g., `A1`). The element in the right side (e.g., `C1`) is eliminated from the full model. Additionally, all the elements that are related with the element in the right (e.g. `D1` and `C2` are related with `C1`) will be related with the element in the left side. In the figure the elements `C1` and `C3` are removed from the model, and the elements `C2` and `D1` are now related with `A1`. Additionally, the element `D2` is now related with `A3`.

More details about the correspondence relationship models, correspondence relationship semantics and correspondence relationship resolution are presented in Chapter 5.

4.2.5 General approach architecture

Figure 4.7 presents the general schema of our approach for adding a *Circle* MTC to the existing *Ellipse* MTC. On the left, the `Ellipse` high-level model

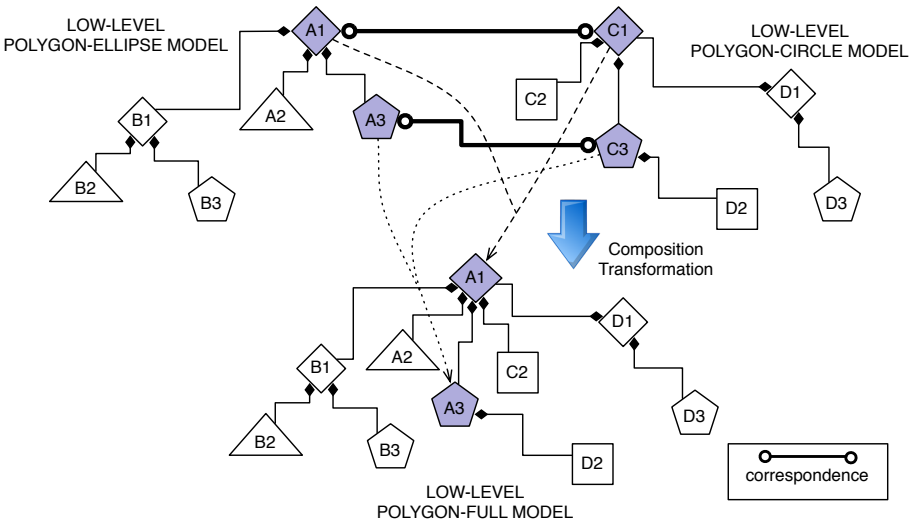


Figure 4.6: Composed model

is transformed into a low-level `Polygon` model. On the right, the `Circle` high-level model is also transformed into a low-level `Polygon` model. The *application modeler* should add manually the $CM_{high-level}$ that is the high-level correspondence model that aligns the two high-level models. $TMellipse$ and $TMcircle$ are the automatically generated tracing models that relate the high-level models with the low-level models. The *MTC modeler* must manually specify the CDM that relates the low-level metamodels with constraints between their metaclasses[2]. The CDM is used to define which elements can be related by a correspondence relationship. The Tcm is an automatically generated transformation that uses the information in the high-level correspondence model $CM_{high-level}$, the tracing models, and the CDM to automatically generate the low-level correspondence model $CM_{low-level}$. Finally, the low-level models are composed and transformed into code by a model-to-text transformation.

4.3 Case Study: Deriving Correspondence Relationships

In Section 4.2 the general idea of our proposed schema was presented with the help of a simple example. This example used geometric shapes as metaphor of metaclasses. The goal of the *geometric* example was to leave out the details of what the models mean and explain our strategy at metamodel level in an abstract and simple way. In the example we introduce the *automatic correspondence derivation mechanism* as the main element of our approach. The

[2]The CDM relates the low-level metamodels and is different from the $CM_{low-level}$ that is automatically generated between the low-level models

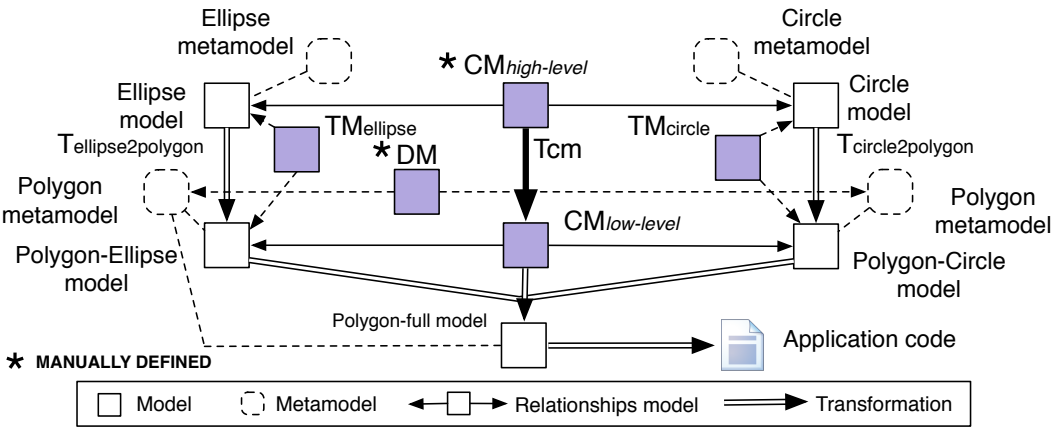


Figure 4.7: General Schema

goal of this mechanism is to provide tools to the modelers in order to align two MTCs. The next sections explain each element of our approach in detail with the help of the case study of the MTC introduced in Section 3.2. The correspondence relationships, their semantics, and resolutions are presented in Chapter 5.

4.3.1 Adding a new concern: *Authorization*

In this section we will briefly reintroduce the example presented in Section 3.2 in order to illustrate in detail our approach at model level. As we previously presented, the MTC that we choose as a case study is used to produce an application named `Risk` by transforming a *Business Model* in several steps into a *Java Model*. Finally, this model is transformed into Java code. The generated applications will offer basic CRUD functionalities such as insert, update, delete, and retrieve the information of business entities. Figure 4.8 presents the original MTC with its metamodels, models, and transformations.

Suppose now that the number of `Risk` users increased and it is required to modify the MTC in order to generate a secure version of it. Specifically, we require to control the user access to `Risk` information and services. The new authorization requirements for `Risk` cannot be specified and implemented using the existing MTC. In order to evolve the existing MTC, we will add an *authorization* specific MTC following the strategy *Parallel Transformation Chain* presented in Section 3.4.5.

4.3.2 The new Security MTC

Having in mind to specify authorization policies without modifying the existing MTC, it is necessary to implement a Security specific MTC. For this, we need

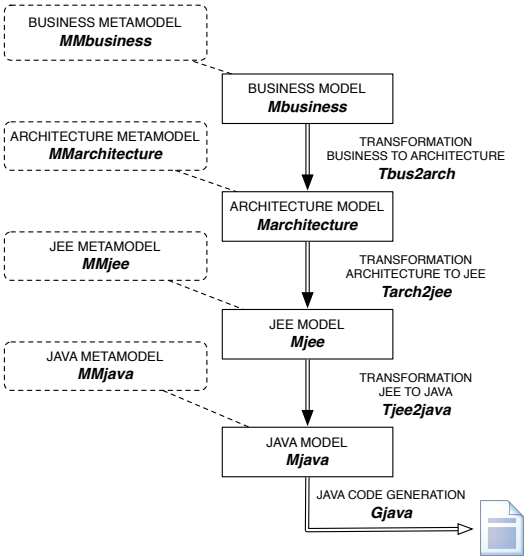


Figure 4.8: The original MTC

to choose or define a metamodel that allows us to specify the new security requirements independently of the existing Business Metamodel. The selected metamodel must be maintained independently from the existing one in order to avoid the *Metamodel pollution* problem presented in Section 1.1.1. With the concepts offered by the metamodel, the new authorization requirements should be specified in a *Security Model*. Similar to the metamodel, this model should be independent of the existing one, in order to avoid the *Monolithic model* problem presented in Section 1.1.1.

High-level Security Metamodel

In our case study, we use the *SecureUML* metamodel presented in [LBD02], which in turn is based on the *Role Based Access Control* (RBAC) model [SCFY96]. The main elements are *Users* and *Groups* that can be assigned to a *Role*. A *Role* can perform a set of *Actions* on a *Resource*. The *Actions* are grouped as a *Permission*.

The original SecureUML goal is to be part of another modeling language, to cover access control aspects [LBD02]. This means that the SecureUML metamodel is an abstract metamodel and it was designed to be extended with the concrete resources and actions that need to be protected. In our case study we extend *Resource* and *Action* to include the resources and the actions that we need to protect. For instance, we extend the concept *Resource* with *ResourceEntity*, *ResourceAttribute* and *ResourceService*. Additionally, the concept *Action* is extended to define every action that can be protected in the concrete resources. Finally, the metamodel is extended to constrain the *Actions*

that can be applied on a concrete *Resource*. For instance, an *ActionExecuteService* can only be applied to a *ResourceService* and not to a *ResourceAttribute*. Figure 4.9 shows the extended metamodel that we call *Security Metamodel*. Additional details of this metamodel are presented in Appendix A.5.

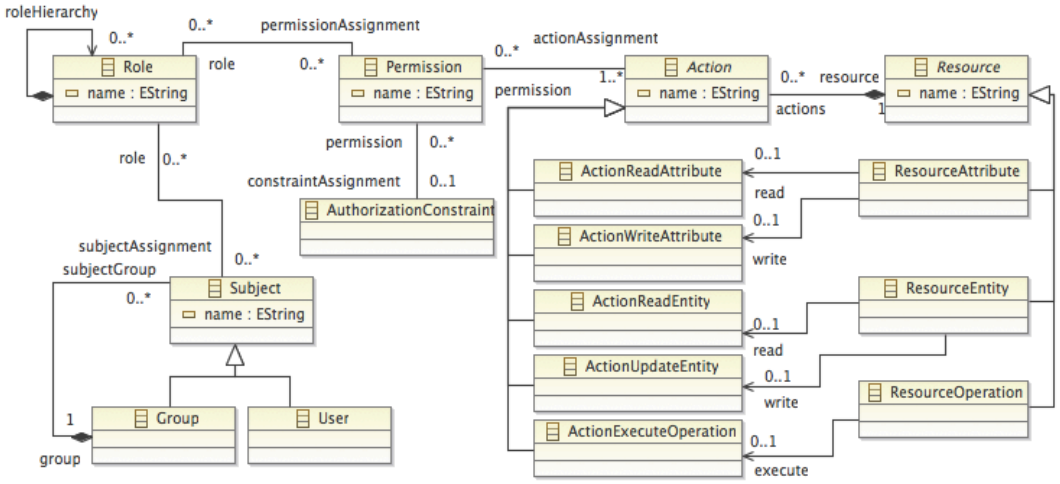


Figure 4.9: High-level Security Metamodel ($MM_{security}$)

4.3.3 High-level Security Model

A fragment of the *Security Model* ($M_{security}$) for `Risk` is presented in Figure 4.10 and it shows two roles: `User` and `Manager`. The *Role* `User` has a permission over `Read` actions on the *ResourceEntity* `Project`. Additionally, the `User` has a permission over `Write` actions on the *ResourceAttribute* `dueDate`. This means that a `User` can read the information about a `Project`, but only can change the project due date. The role `Manager` inherits `User` permissions and adds *Write* actions on the `Project`. In summary, the `Manager` has read and write *Permissions* on the information of a `Project`.

Low-level Java Metamodel ($MM_{security-java}$)

We use as a target low-level for security the same metamodel used by the existing MTC: the *Java Metamodel*. As presented in Section 3.2.6, this metamodel is based on the *J2SE5 Metamodel* of the MoDisco project (`http://wiki.eclipse.org/MoDisco/J2SE5`). This metamodel is the reflection of the Java language and some parts of it are presented in Appendix A.4. This metamodel contains concepts such as *ClassDeclaration*, *MethodDeclaration*, *FieldDeclaration*, etc.

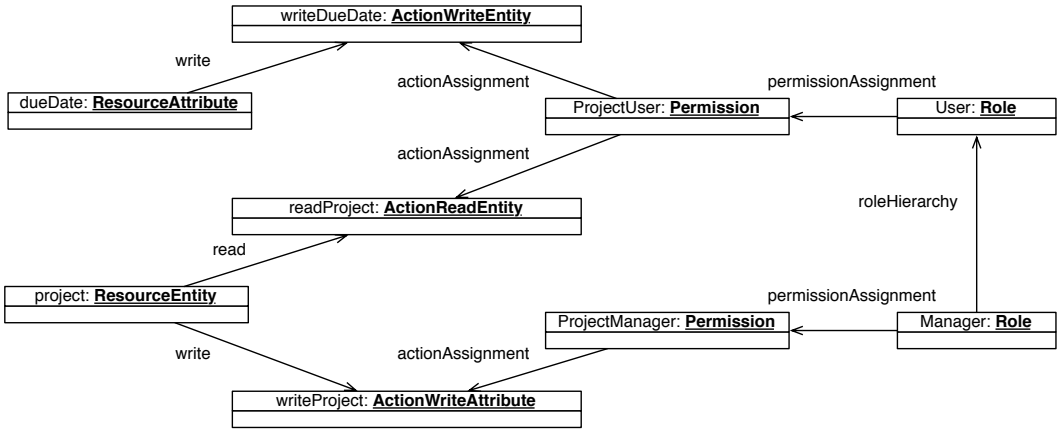


Figure 4.10: High-level Security Model ($M_{security}$)

Security to Java MTC ($T_{sec2java}$)

When the source and target metamodels are selected, it is necessary to implement the MTC that is going to transform high-level security models into Java Models. In this case we only use one transformation step that translates the authorization concepts into Java concepts. This model-to-model transformation is called $T_{sec2java}$[3]. $T_{sec2java}$ transforms a high-level security *Resource* into a secured *Class*, *MethodDeclaration*, or *Field*. An *Action* is transformed into an annotated *MethodDeclaration*, and *Roles* are added as *Properties* in the methods *Annotations*. For example, the *AttributeResource* `dueDate` is transformed into a `private` *Field*, the action/permission *ActionWriteAttribute* on `dueDate` is transformed into an annotated method *MethodDeclaration* and the *Role* `Manager` into a *Property* of the *Annotation*. The *Annotation* `@RolesAllowed` is the mechanism used by JEE to offer access control [CS09].

4.3.4 High-level Correspondence Model ($CM_{high-level}$)

When the new MTC is built and the new high-level Security model is specified, it is necessary to identify the corresponding elements. We align the two high-level models using a CM which relates the elements that represent the same "thing" in both domains (e.g., business and security domains). For example, in Figure 4.12 the *Business Model* ($M_{business}$) contains the *BusinessEntity* `Project` and the *Security Model* ($M_{security}$) contains the *ResourceEntity* `Project` that needs to be protected. The CM between both models $CM_{high-level}$ is represented in Figure 4.12 by black lines with circles at the ends. In this chapter we are not going to explain the semantics of the correspon-

[3]The code of this transformation is available at http://qualdev.uniandes.edu.co/mtcframework.

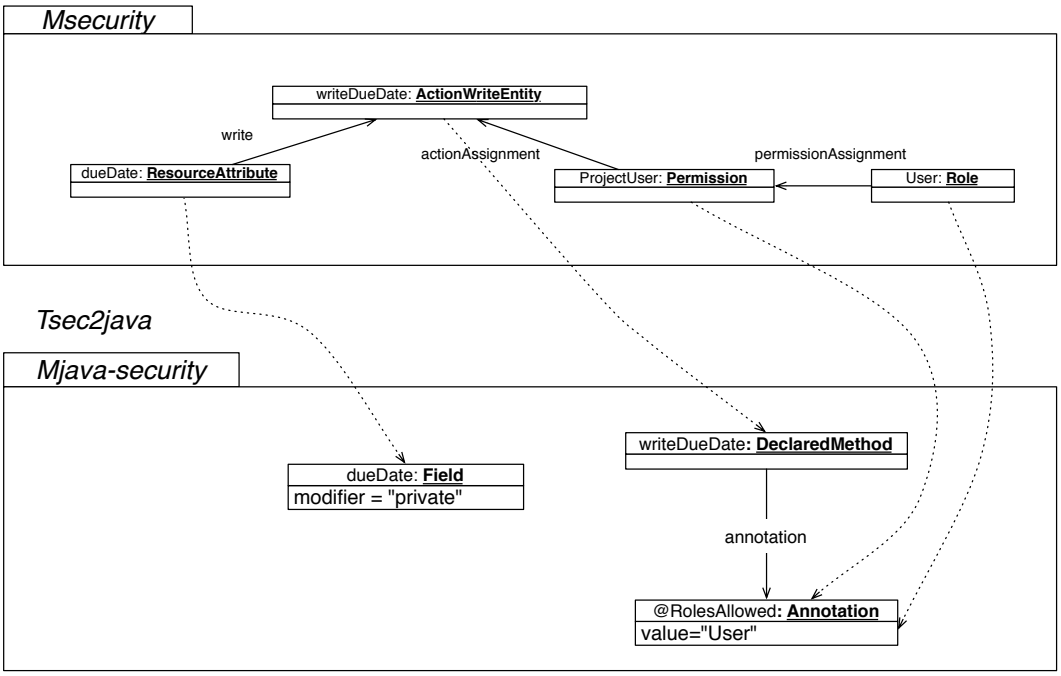


Figure 4.11: Security to Java Transformation ($T_{sec2java}$)

dences, this is going to be explained in Chapter 5. The $CM_{high-level}$ contains a correspondence relationship with links to the *BusinessEntity* `Project` and the *ResourceEntity* `Project`. This relationship means that the *ResourceEntity* `Project` and the *BusinesEntity* `Project` represent the same "thing" in two different domains. Additionally, the *Attribute* `dueDate` in the Business model is related in the CM to the *ResourceAttribute* `dueDate` in the security model. The modeler manually creates these correspondence links because he understands the meaning of the relationships between elements. Although the corresponding elements are "cloned" in both models, only the information relevant to each domain is copied. For instance, the structural relationships between resources is not important in the security model, therefore there is no association between the *ResourceEntity* `Project` and the *ResourceAttribute* `dueDate`.

4.3.5 Low-level models

In the presented example, the two low-level models (M_{java} and $M_{java-security}$) are partial complementary specifications of the required application. These two models need to be statically integrated into one single model in order to obtain the complete Java application. As explained in Section 2.4.2, in order to integrate two models we need to identify *what* to compose, i.e., which elements in the models have to be composed [JFB08]. Although in some models it is

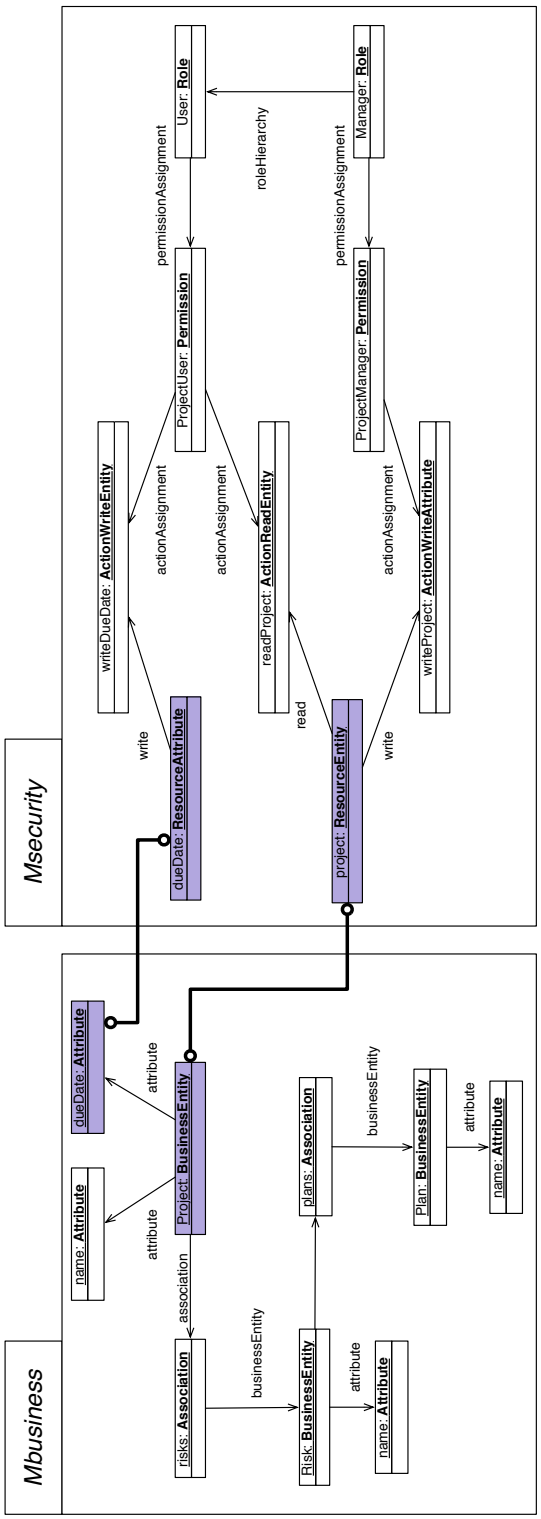


Figure 4.12: High-level Correspondence Model

possible to use heuristics to identify corresponding elements (e.g., elements with the same name), we choose to manually create the CM because sometimes the semantic differences between the metamodels is too high and only an expert modeler can create the CM. Correspondence Models are presented in the next chapter.

Once, the CM model between the high-level models is constructed manually by the modeler it is time to obtain the CM model between the low-level models. At the low-level of abstraction we use our proposed Correspondence Derivation Mechanism (CDM) to identify which are the corresponding elements. The end result is that the CDM will generate the CM between the low-level models. This is a key characteristic of our approach because it frees the *application modeler* from working at implementation level and fulfill one of our goals of working at high-level of abstraction.

On the left side, Figure 4.13 presents the transformation of the *Business Model* $M_{business}$ into a *Java Model* M_{java}. On the right side, Figure 4.13 presents the transformation of the *Security Model* $M_{security}$ into a *Java Model* $M_{java-security}$[4]. For now, suppose that the *Attribute* `dueDate` is only transformed into the *FieldDeclaration* `dueDate` and its setter *MethodDeclaration* `setDueDate`.

On the bottom, Figure 4.13 shows that there are some elements (i.e., colored elements) that are generated from high-elements that have a correspondence relationship between them. As presented in the *geometrical* example, this intuitively means that they are corresponding elements, but we need to identify the "correct" matches. In the next section, the correspondence derivation process is presented in detail.

4.4 Derivation Requirements

In the previous section the new MTC was implemented with the purpose of transforming a high-level security model into a partial Java implementation. Additionally, a CM was defined between the high-level models identifying the corresponding elements. In this section, we define the two conditions required to create a correspondence relationship between two target elements.

On the one hand, in Section 4.2.2 we intuitively introduced the necessity of tracing back the elements in the low-level models and to verify if they originate from a pair of elements related by a correspondence relationship. On the other hand, in Section 4.2.3 we intuitively introduced that we need to constrain the possible correspondence relationships that can be created between the target elements. Initially, these constraints are only based on the metaclasses that the target elements conform to.

[4]In this figure only the transformation of the *Attribute* `dueDate` is presented to illustrate our approach.

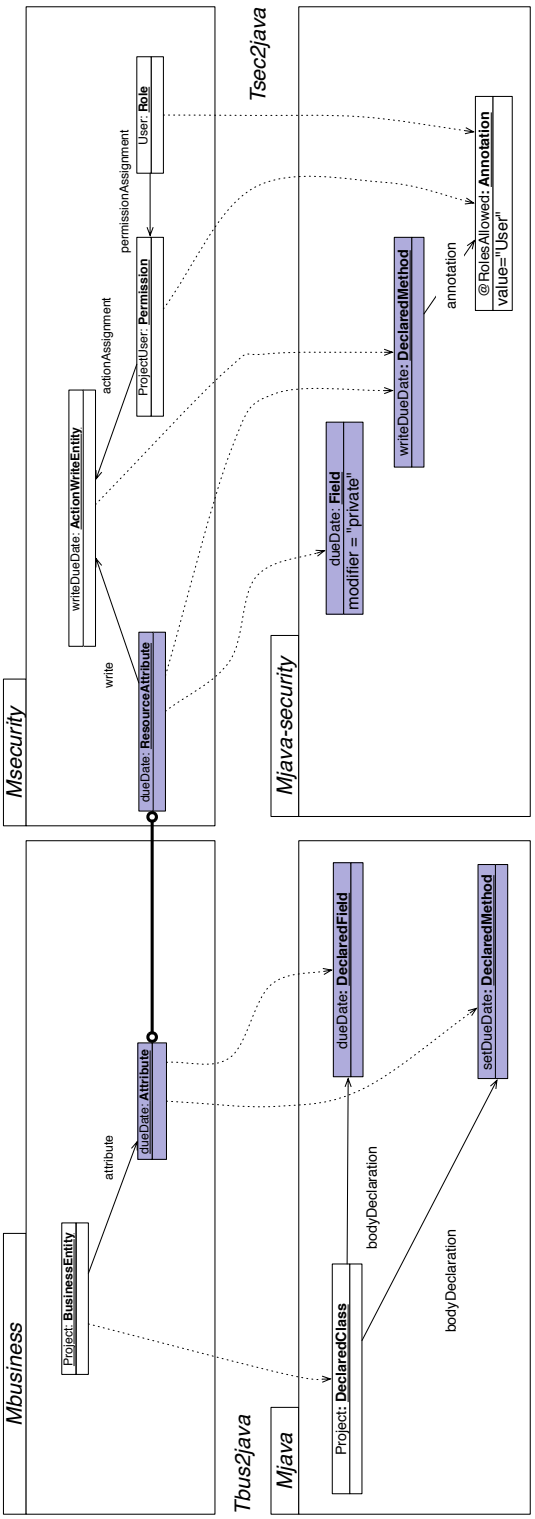


Figure 4.13: Business 2 Java & Security 2 Java

Generalizing these two conditions, we can state: two elements a' and b', from the M_A and M_B models respectively, will have a correspondence relationship if they fulfill two conditions:

- **Tracing back targets that are generated from corresponding sources**
  There is a CM relationship at the higher level between a and b, from M_A and M_B models respectively, where a' was generated from a by the transformation T_1, and b' was generated from b by the transformation T_2.

- **Constraining the generated correspondence relationships**
  The metaclasses ma' and mb' where a' conforms to ma' and b' conforms to mb' have a constraint that *allows* for correspondence relationship between their instances. If a constraint does not exist between the metaclasses this condition is not fulfilled.

Intuitively, the first condition establishes that elements a' and b' trace back to a pair of elements that have a high-level correspondence relationship between them. The second condition means that the metaclasses ma' and mb' are the same metaclass or extensions of the same one. Therefore, it is permitted to define correspondence relationships between their instances. If both conditions are satisfied for a pair of elements a' and b', the *Correspondence Derivation Mechanism* will produce a correspondence relationship between a' and b'. On the contrary, a corresponding relationship between two elements c' and d', which trace back to a and b respectively, cannot be created if there is no constraint that allows corresponding relationships between their metaclasses mc' and md'.

Figure 4.14 shows the correspondence derivation schema. In the figure a correspondence relationship is created by the correspondence derivation mechanism between a' and b' because the two conditions are fulfilled. However, the elements c' and d' do not fulfill the constraints. Hence, no relationship is created. In this figure we use an "allowed" relation between the metaclasses ma' and mb' to intuitively introduce the idea of derivation constraints. The derivation constraints and the derivation metamodel are explained in Section 4.6.

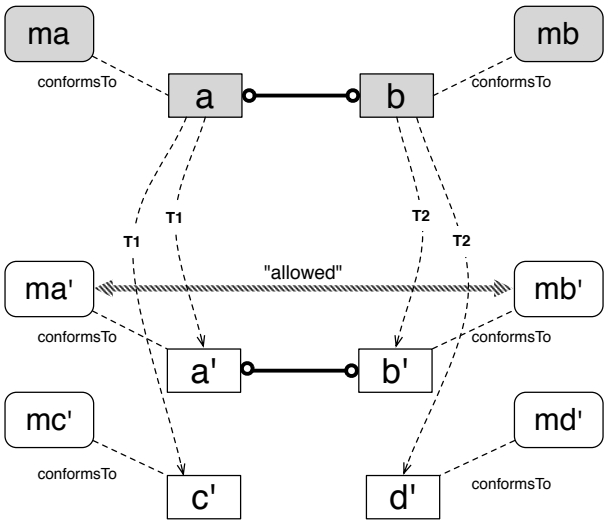


Figure 4.14: Correspondence Derivation schema

4.5 Tracing back corresponding elements

As explained intuitively in Section 4.2.2 it is required to identify the low-level elements that are generated from a pair of high-level corresponding elements. A *Tracing Model* is the mechanism needed in order to identify the sources for each target low-level element.

4.5.1 Tracing Metamodel

A *Tracing Metamodel* should offer concepts that allow the representation of tracing links between the source and target elements for each transformation. A Tracing metamodel should at least have a concept that represents the *root* of the tracing model and a concept that represents the tracing relationships. Each tracing relationship should identify the source elements and the target elements that were generated by the transformation.

In the implementation of the correspondence derivation mechanism we use the Tracing metamodel presented in Figure 4.15. This metamodel was developed as part of an extension of the *ATL Virtual Machine* (ATL-VM). This extension is presented in Section 6.5 and allows to automatically generate tracing models when an ATL transformation is executed. This metamodel was developed for the ATL-VM, but it can be used to generate tracing models with any EMF based transformation engine, such as OpenArchitecture [ope10a].

The Tracing metamodel has a *TransientLinkSet*[5] metaclass that represents the *root* of the tracing model, which is a collection of tracing links. The

[5]The name of the metaclasses of the Tracing metamodel are selected in order to have compatibility with the internal types of the ATL-VM.

TransientLink metaclass represents a tracing link. Each tracing link has a collection of *sourceElements* and a collection of *targetElements*. Finally, this metamodel has a *TransientElement* metaclass that represents traced elements. A *TransientElement* has the attribute *name* and a reference called *value*. This reference points to the actual elements in the source and target models.

4.5.2 Generating tracing models

There are several options to generate a tracing model when a transformation is executed:

1. The first strategy is to manually add the tracing logic to the existing transformation rules.

2. The second option is to use a *High-Order Transformation* (HOT). In [Jou05], a HOT is presented to automatically append the additional output elements to the original rules in order to generate a required tracing model. Nevertheless, this solution creates an overhead problem because every time that the original transformation rules are changed, it is necessary to re-apply the HOT.

3. The final option depends on the functionality of the transformation engine. As we mentioned before, we extend the ATL-VM to automatically generate tracing models.

For instance, when a transformation is applied to the *Attribute* `dueDate` in the *Business model*, it is transformed into the *FieldDeclaration* `dueDate` and the *MethodDeclaration* `setDueDate`. In order to make this information available for the derivation mechanism, we generate tracing links between target elements and source elements. The same happens on the security side, where the derivation mechanism needs to know if the *MethodDeclaration* `writeDueDate` traces back to a related *ResourceAttribute*. Once both transformations are executed, two tracing models are generated (TM_{bus} and TM_{sec}). With these two tracing models, the derivation mechanism can find the elements in both lower-level models that trace back to the pair of related elements in both higher-level models. We generate a tracing model when each transformation is executed. This model conforms to a Tracing Metamodel presented before, and it has links between every source element and its target elements.

4.5.3 Composing tracing models

When several transformation steps are used in a MTC, each transformation step produces a Tracing Model. In order to find if an element of the lowest level model comes from a corresponding element in the high-level model, we need to

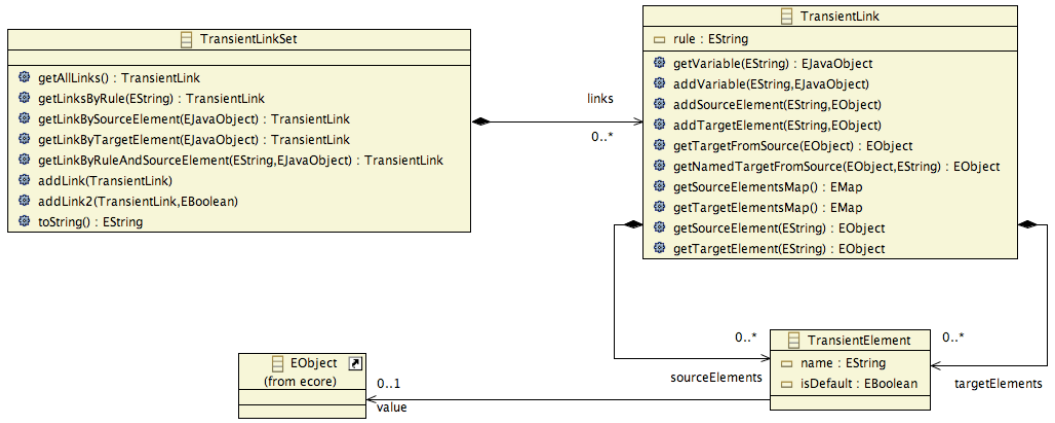


Figure 4.15: Tracing Metamodel

compose all the traces. In other words, we need to obtain the *transitive closure* for each target element. Therefore, to trace back from the lowest level model to the highest one, all the TMs generated by the MTC must be composed into a single tracing model that relates the highest level model with the lowest one.

For instance, in the MTC presented in Section 3.2, three transformations are executed. Figure 4.16 shows how each transformation produces a Tracing model: 1) the tracing model `TMbus2arch` that relates the *Business model* with the *Architecture model*, 2) the tracing model `TMarch2jee` that relates the *Architecture model* with the *JEE model*, and 3) the tracing model `TMjee2java` that relates the *JEE model* with the *Java model*.

Keeping in mind the need of identifying the correspondences in the lowest level models, one needs to verify which elements are generated from corresponding high-level elements. For this reason, the three tracing models must be composed. Figure 4.17 shows how these tracing models are composed[6].

The first step is to compose the first two models in order to obtain the tracing model `TMbus2jee` (Figure 4.17(b)) that relates the *Business model* with the *JEE model*. This composition is performed by identifying the source elements in `TMarch2jee` that are the same as the target elements of `TMbus2arch`. In the Figure 4.17(a) the tracing relationship `T1` has the same element as target as the sources of the tracing relationships `T2` and `T3`. For each one of the elements that match the target and source element, a new tracing relationship is created between the source element of the first model and the targets in the second model. In Figure 4.17(b), the TA tracing link is created between the source of `T1` and the target of `T2`. The ATL transformation that we use to compose two tracing models is presented in Appendix B.1.

The second step is to compose the third TM model `TMjee2java` with the new tracing model `TMbus2jee` that relates the *Business model* with the *JEE*

[6]The relationship between the models elements are excluded for simplicity

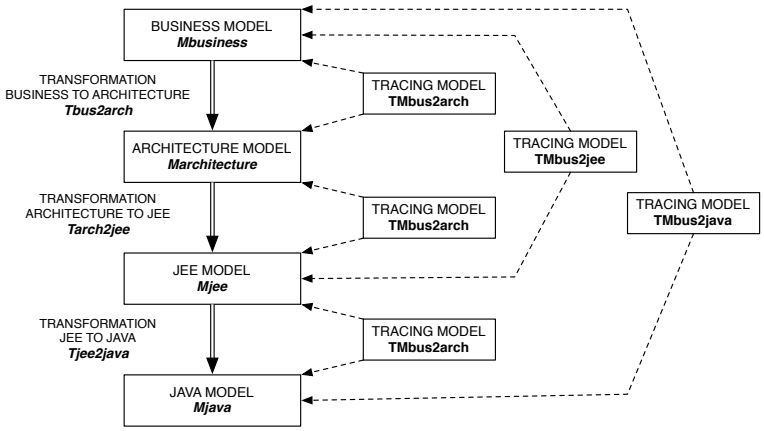


Figure 4.16: Tracing composition

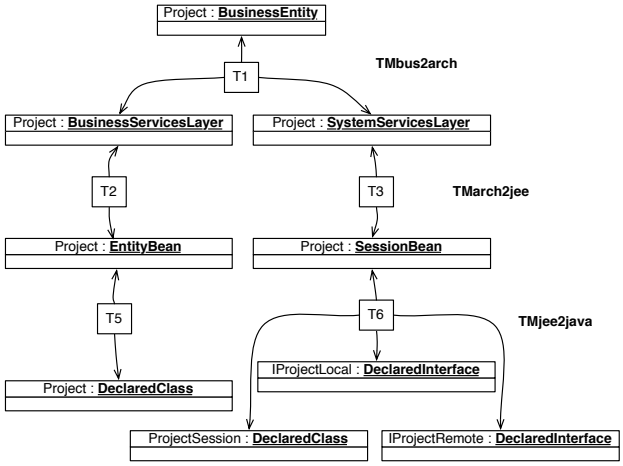
model. The result of this composition is a tracing model `TMbus2java` that relates the *Business model* with the *Java model*. This composition is performed identifying the source elements in `TMjee2java` that are the same as the target elements of `TMbus2jee`. Next a new tracing relationship is created between the source element of the first model and the targets in the second model. This model can be used by the correspondence derivation mechanism to identify which low-level target elements are generated from a pair of corresponding high-level elements.

Although composing the tracing models allows us to easily identify if the source of a target element has a correspondence relationship, we loose intermediate information. The reason for this is the lack of dedicated-tracing operations to navigate through the different tracing steps involved in a transformation chain in ATL. However, this information is only useful when is not possible to differentiate between the target elements of a single source element (i.e., when two or more target elements conform the same metaclass and share the same trace name). When this happens, the tracing composer transformation will inform us of the problem. This is going to be explained in detail in Section 4.7.

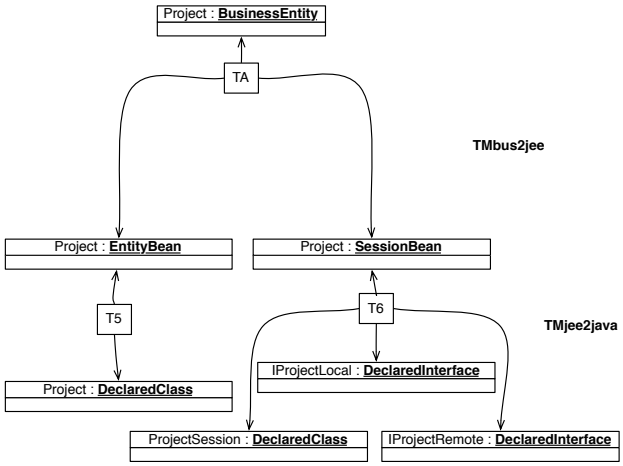
4.6 Constraining the correspondence relationships

As presented in Section 4.2.3, once we identify the elements that are generated from a couple of corresponding high-level elements, we need to verify if it is possible to define a correspondence between them. For instance, we have two corresponding high-level elements `a` and `b`, and the MTCs `T1` and `T2`. The MTC `T1` transforms `a` into `a1` and the MTC `T2` transforms `b` into `b1`. It is obvious that a relationship exists between `a1` and `b1` (Figure 4.18(a)).

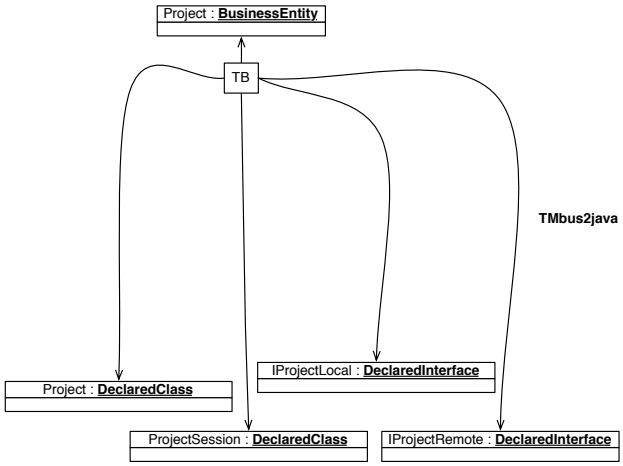
However, Figure 4.18(b) presents a slightly more complex case. In this case



(a)



(b)



(c)

Figure 4.17: Composing tracing models

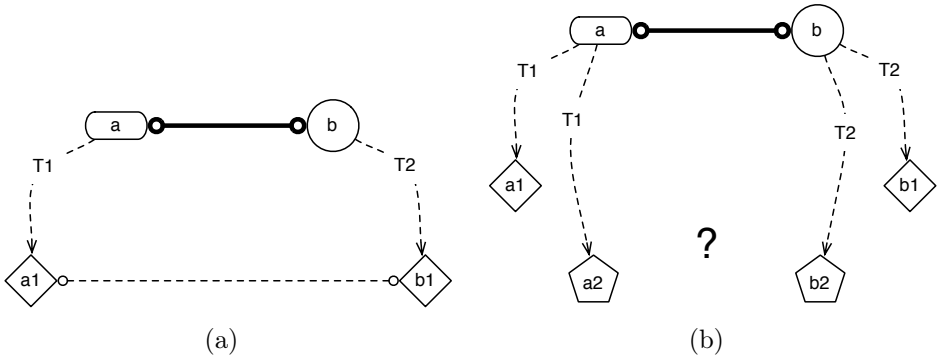


Figure 4.18: Constraining the correspondence relationships

the MTC `T1` transforms `a` into `a1` and `a2` and the MTC `T2` transforms `b` into `b1` and `b2`. In this case, it is possible to define four different relationships between the elements (i.e., `a1` and `b1`, `a1` and `b2`, `a2` and `b1`, or `a2` and `b2`). Therefore, only with the tracing information it is not enough to identify which are the correct relationships to generate.

In order to constrain the possible correspondence relationships that can be generated between the target elements, we define a *Correspondence Derivation Model* (CDM), in which we use the metaclass information to identify the correct match. In Figure 4.18 the metaclass information is represented by the geometric shapes (i.e., ellipse, circle, diamond, pentagon). For instance, the only valid relationships that can be defined are between elements that conform to the same metaclass (i.e., with the same shape). In the Figure 4.18(b) the only valid relationships are between `a1` and `b1`, and `a2` and `b2`.

4.6.1 Correspondence Derivation Model

The *Correspondence Derivation Model* (CDM) is a key element in the generation of the low-level correspondence model. Using this model, we specify constraints that allow or restrict the generation of correspondence relationships between the low-level elements.

The CDM explicitly defines if a correspondence relationship can be defined between the instances of two metaclasses, in two metamodels. The related metamodels can be the same when the target platform (e.g., Java) is the same or different when the target platforms are different (e.g., Java and SQL).

If it is possible to define a correspondence relationship between the two elements, we state that they conform to *compatible* metaclasses. The *MTC developer* must decide if two metaclasses are compatible or not. Similarly, the modeler must decide about the propagation of the compatibility relationships. If two metaclasses are compatible, he needs to decide if their submetaclassess or their composites are compatibles as well. Whether or not correspondence

relationships are allowed between certain model elements is defined using *compatibility constraints*.

MTC developers must define a CDM between the target metamodels to make explicit if the instances of two metaclasses are allowed to be related by a correspondence relationship or not. Figure 4.19 shows a fragment of the CDM specified with the Java Metamodel (MM_{java}). In this case, the Java metamodel is the target metamodel for both MTCs and the compatibility constraints must be defined between its metaclasses. For instance, in Figure 4.19 a *CompatibleLink* is defined between the metaclass *ClassDeclaration* and the metaclass *ClassDeclaration*, meaning that it is possible to generate a correspondence relationship between the classes that conform to them. In other words, if two metaclasses come from the same source they are going to be composed. There are *CompatibleLinks* between *DeclaredPackages*, *DeclaredInterfaces MethodDeclarations* and *FieldDeclarations* as well. These are the elements that are compatibles and we expect to compose in order to produce the full application.

Moreover, in Figure 4.19 there is one *CompositionLink* between the relationship *ownedElements* (*PackageDeclaration* and *AbstractTypeDeclaration*) and the *AbstractTypeDeclaration*. This relationship will allow to propagate correspondence relationships from two corresponding packages to their owned elements (e.g., Classes and Interfaces). Similarly, a *CompositionLink* is defined between the relationship *bodyDeclarations* (*AbstractTypeDeclaration* and *BodyDeclarations*) and *BodyDeclarations* in order to propagate the correspondence relationships to any method and field declared in a class.

In summary, the developer must decide about the propagation of the compatibility relationships. It is possible to say that in the *Correspondence Derivation Model* the *MTC developer* specifies intensional formulas to derive the low-level correspondence model. How a correspondence relationship is propagated is specified by using different types of constraints, that are defined in the *Correspondence Derivation Metamodel*.

4.6.2 Correspondence Derivation Metamodel

Compatibility relationships represent constraints that allow or deny the creation of a correspondence relationship between target models. We have defined different types of compatibility relationships in the *Correspondence Derivation Metamodel* (CDMM) which offers the developer the possibility of specifying how a low-level correspondence relationship between two elements in the target models is propagated, provided that they come from two elements related by a high-level correspondence relationship. Figure 4.20 presents the CDMM. The base concept of the metamodel is the *DerivationLink* that relates two metaclasses. The *DerivationLink* is an abstract concept that has several specializations representing the types of constraints: *CompatibleLink, FinalLink,*

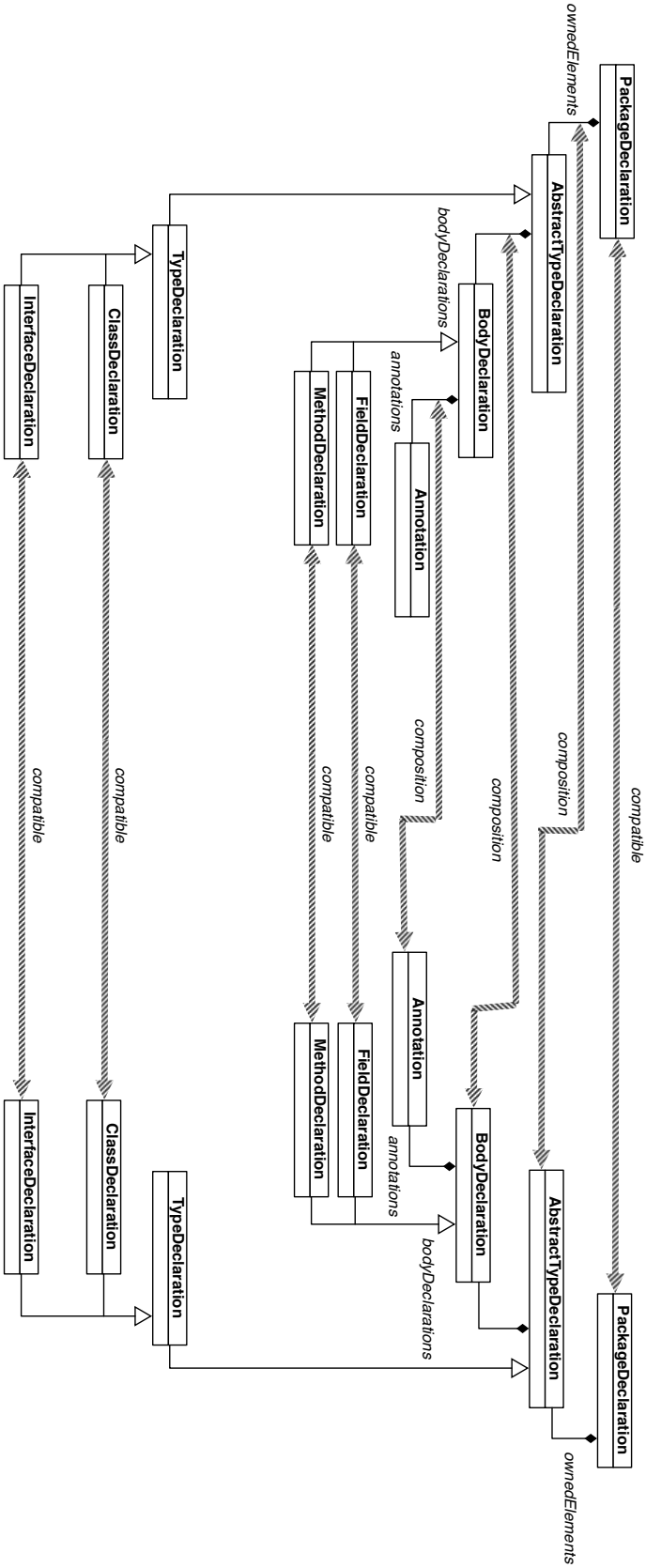


Figure 4.19: MM_{java} - MM_{java} Correspondence Derivation Model

IncompatibleLink and *CompositionLink*. The *DerivationLink* metaclass contains the attribute *targetType* which is the type of the correspondence relationship that must be created. The types of target relationships are presented in the next chapter. Finally, the metamodel has the metaclass *DerivationElement* that represents a metaclass in a metamodel. This metaclass has a reference to an *EObject* that allows us to relate the CDMM to any *Ecore* based metamodel. The explanation of each type of *DerivationLink* is presented in the next subsections.

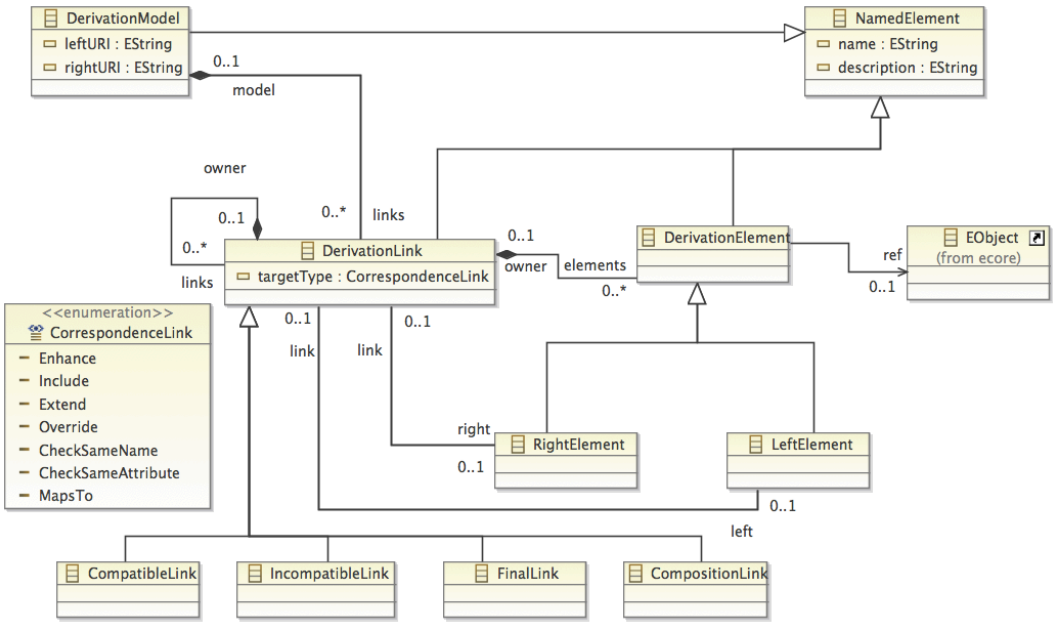


Figure 4.20: Derivation Metamodel

4.6.3 Compatible Link

Figure 4.21 shows the *Compatible link* derivation schema. A *Compatible link* relates two metaclasses *ma'* and *mb'*, each one in a different metamodel. In the figure, a correspondence relationship is created between a' and b' because:

- a' and b' are generated from a and b in the higher-level models.
- a and b are connected by a high-level correspondence relationship.
- a' and b' conform to the metaclasses ma' and mb' respectively.
- there is a *Compatible link* between ma' and mb'.

Additionally, a correspondence relationship is created between c' and d' because:

- c' and d' are generated from a and b in the higher-level models.
- a and b are connected by a high-level correspondence relationship.
- c' and d' conform to the metaclasses mc' and md' respectively.
- the metaclasses mc' and md' are submetaclasses of ma' and mb' respectively.
- there is a *Compatible link* between ma' and mb' that can be propagated to its subclasses.

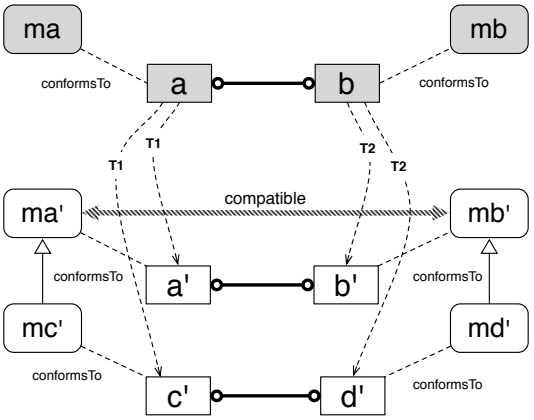


Figure 4.21: Compatible Link

For instance, a *Compatible link* is used between the metaclass *FieldDeclaration* in $MMjava$ and the metaclass *FieldDeclaration* in $MMjava-security$ will be related with a *CompatibleLink*, because we want to protect every *FieldDeclaration* in the application. Therefore, an instance of *FieldDeclaration* or an instance of one of its submetaclasses will be private.

4.6.4 Final link

Figure 4.22 shows the *Final link* derivation schema. A *Final link* relates two metaclasses *ma'* and *mb'*, each one in a different metamodel. Similar to the `Compatible Link` a correspondence relationship is created between a' and b'. However, a correspondence relationship cannot be created between c' and d' because:

- c' and d' are generated from a and b in the higher-level models.
- a and b are connected by a high-level correspondence relationship.
- c' and d' conform to the metaclasses mc' and md' respectively.
- the metaclasses mc' and md' are submetaclasses of ma' and mb' respectively.
- there is a *Final link* between ma' and mb' that **cannot** be propagated to its subclasses.

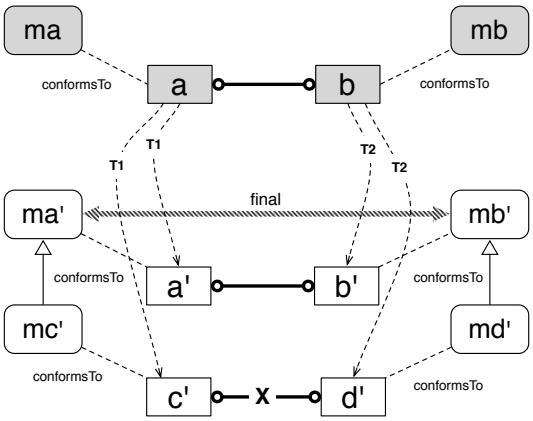


Figure 4.22: Final Link

For instance, a *Final link* is used between the metaclass *MethodDeclaration* in $MMjava'$ and the metaclass *MethodDeclaration* in $MMjava-security$ will be related with a *FinalLink*, because we want to protect only the method types and not their subclasses. If we use a *CompatibleLink* for these metaclasses and there are subclasses of them, then the instances of those subclasses will be related as well.

4.6.5 Incompatible Link

Figure 4.23 shows the *Incompatible link* derivation schema. An *Incompatible link* relates two metaclasses *mc'* and *mb'*, each one in a different metamodel. Similar to the *Compatible Link*, a correspondence relationship is created between a' and b'. However, a correspondence relationship cannot be created between c' and d' because:

- c' and d' are generated from a and b in the higher-level models.
- a and b are connected by the a high-level correspondence relationship.
- c' and d' conform to the metaclasses mc' and md' respectively.
- the metaclasses mc' and md' are submetaclasses of ma' and mb' respectively.
- there is an *Incompatible link* between mc' and md' that explicitly forbids to create correspondence relationships between the their instances.

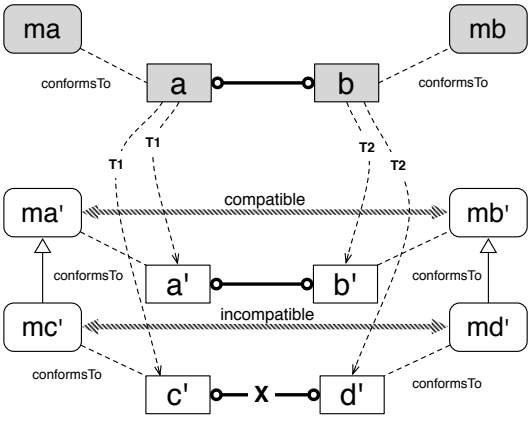


Figure 4.23: Incompatible Link

This type of link is used to *explicitly* block the propagation of *CompatibleLinks* and *CompositionLinks* in some specific submetaclasses or composite metaclasses respectively.

4.6.6 Composition Link

Figure 4.24 shows the *Composition link* derivation schema. A *Composition link* relates two composition associations r_a and r_b, each one in a different metamodel and relating two different metaclasses in each metamodel. In the figure, a correspondence relationship is created between c' and d' because:

- a' and b' are generated from a and b in the higher-level models.
- a and b are connected by a high-level correspondence relationship.
- c' and d' are not generated from a and b in the higher-level models.
- a' and b' conform to the metaclasses ma' and mb' respectively.
- c' and d' conform to the metaclasses mc' and md' respectively.
- r_a is a containment relationship between the metaclasses ma' and mc'.
- r_b is a containment relationship between the metaclasses mb' and md'.
- there is a *Composition link* between r_a and r_b.

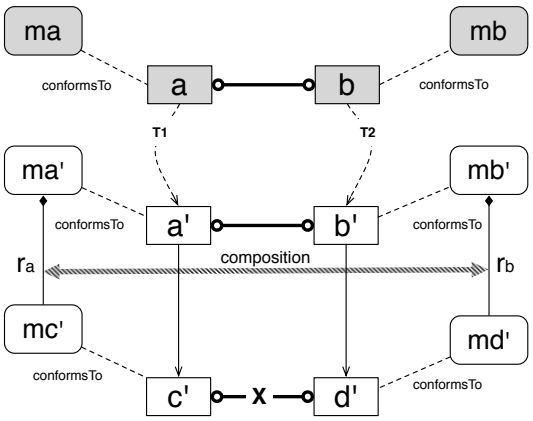


Figure 4.24: Composition Link

For instance, with a *CompositionLink* between the containment *ClassDeclaration/FieldDeclaration* relationship, we can explicitly express that if a *ClassDeclaration* is protected, then every contained *FieldDeclaration* is protected as well.

4.6.7 Generating the Correspondence Model Transformation

Once a high-level correspondence model is specified, the *tracing models* are obtained, and the *Correspondence Derivation Model* is defined by the *MTC*

developer, they need to be processed by a transformation. This transformation that we call *Correspondence Model Transformation* (CMT) is responsible for analyzing the information of the models in order to generate the low-level correspondence model. However, the CDM is defined at metamodel level and the CMT needs information at model level. Therefore, it is necessary to express the constraints as part of the transformation rules. For this reason, we use a *high-order transformation* that receives the CDM as input and produces the CMT. As we explained before, a HOT is a transformation rule that produces another transformation rule as output. The HOT that we implemented uses the CDM as input and adds a new rule to CMT for each constraint specified in the CDM. The added rules are in charge of selecting and connecting the elements depending on the constraint in the CDM and the tracing information. For instance, a *CompatibleLink* is defined between the metaclass *MethodDeclaration* in MM_{java} and the metaclass *MethodDeclaration* in MM_{java}. The semantics of this constraint allows the derivation of correspondence relationships between elements that conform to the related metaclasses or their submetaclasses. Listing 4.1 presents part of the HOT rule that we implemented in ATL.

```
1  rule CompatibleLink2Rule {
2    from
3      inlink : DerivationMM!CompatibleLink
4      to
5        outrule : ATL!MatchedRule (
6          name <- 'CompatibleLink' + inlink.left.name + inlink.right.name,
7          inPattern <- inPattern,
8          outPattern  <- outPattern
9        ),
10       inPattern : ATL!InPattern (
11         elements  <- Sequence{leftInElement, rightInElement, link},
12         filter    <- derivationConstraint
13       ),
14       derivationConstraint : ATL!OperationCallExp ( ... ),
15       ...
16       outPattern : ATL!OutPattern (
17         elements <- Sequence{outLink}
18         ...
19       ),
20       outLink : ATL!SimpleOutPatternElement (
21         varName <- 'outlink',
22         bindings <- Sequence {linkName, linkModel, leftElement, righElement}
23         ...
24       ),
25       ...
26     }
27 ...
```

Listing 4.1: High Order Transformation

The main goal of the presented HOT is to produce an ATL *MatchedRule* for each *Derivation Constraint* specified in the DM. For instance, the rule takes the *CompatibleConstraint* as input (line 3) and produces a new *MatchedRule* (line 5-9). An *inPattern* and an *outPattern* are defined for the new *MatchedRule*. The *inPattern* (lines 10-13) specifies the type of the elements that are going

to be matched by the rule, such as *MethodDeclaration* and *MethodDeclaration*. Additionally, the *inPattern* has a filter where the derivation constraint is defined. In the *derivationConstraint* (line 14) an intensional formula is specified based on the type of constraint. These formulas are expressed as OCL constraints that are applied as a filter to the matched elements. The *outPattern* (lines 16-19) specifies the output elements that will be generated. In this case a new correspondence link between the matched elements.

Using an ATL HOT to generate the CMT allows to align an arbitrary pair of complementary MTCs. The requirements as presented before are: 1) both MTCs must generate tracing models, 2) the Correspondence Derivation Model between the target metamodels must be defined by the *MTC developer*.

4.6.8 The Correspondence Model Transformation (CMT)

When the HOT presented in Section 4.6.7 is executed, the Correspondence Model Transformation is generated. As explained before, the HOT processes the CDM and produces a rule for each specified constraint. The CMT is responsible for generating the low-level correspondence model. For instance, one of the rules generated from the CDM presented in Figure 4.19 is presented in Listing 4.2. This transformation rule takes three elements as input: 1) an element of type *MethodDeclaration* (line 3) from the $Mjava$ model, 2) an element of type *MethodDeclaration* (line 4) from the $Mjava-security$, and 3) a correspondence link that relates elements which are the sources for (1) and (2) (lines 5-10). This rule has three outputs: 4) a correspondence link (lines 12-16), 5) a reference to the left element that is the left input element (1) (line 17) and 6) a reference to the right element that is the right input element (2) (line 18). The types of the input and output correspondence links are explained in Chapter 5. The complete generated rule is presented in Appendix B.2.

As presented in Listing 4.2, the CMT is specified for both low-level metamodels (lines 3-4), and the types of the metaclasses are hard-coded in the inputs of the rule; the metaclass information comes from the CDM constraints because we automatically generate CMT using the presented HOT. In our case study, the *MethodDeclaration* `setDueDate` at the application side can be connected to the *MethodDeclaration* `writeDueDate`, because the metaclasses *MethodDeclaration* and *MethodDeclaration* are compatible. As we presented before, a constraint exists that allows these two elements to be connected. This makes it possible to relate only the compatible pairs of elements and not every generated element in each side.

```
1 rule EnhanceLink_MethodDeclaration_MethodDeclaration {
2   from
3     leftElement : LEFTOUT!MethodDeclaration,
4     rightElement : RIGHTOUT!MethodDeclaration,
5       inlink : CORRESPONDENCE!CompositionLink(
6         (inlink.left.getTargets('leftTrace')->flatten()->select(e |
7           e = leftElement)->notEmpty()) and
8         (inlink.right.getTargets('rightTrace')->flatten()->select(e |
9           e = rightElement)->notEmpty())
10      )
11  to
12    outlink : CORRESPONDENCE!EnhanceLink (
13      model <- thisModule.matchModel,
14      left <- leftEnd,
15      right <- rightEnd
16    ),
17    leftEnd : CORRESPONDENCE!LeftElement (ref <- leftElement),
18    rightEnd : CORRESPONDENCE!RightElement (ref <- rightElement)
19  }
20 }
```

Listing 4.2: Correspondence Model Transformation (CMT)

4.7 Extending the scope of the derivation mechanism

The use of relationships between the target metamodels is sufficient only when every target element generated from a single source element is an instance of a different metaclass. However, usually MTCs transform a single source into several elements that conform to the same metaclass. Figure 4.25 shows on the one hand how the MTC `Tbus2java` transforms the *Attribute* `dueDate` from the Business model into the *FieldDeclaration* `dueDate`, the setter *MethodDeclaration* `setDueDate` and the getter *MethodDeclaration* `getDueDate`. On the other hand the MTC `Tsec2java` transforms the *ResourceAttribute* `dueDate` from the Security model into a private *FieldDeclaration* `dueDate`, and the annotated *MethodDeclaration* `writeDueDate`.

Therefore, these five target elements (i.e., the *FieldDeclaration* `dueDate`, the setter *MethodDeclaration* `setDueDate`, the getter *MethodDeclaration* `getDueDate`, the private *FieldDeclaration* `dueDate`, and the annotated *MethodDeclaration* `writeDueDate`) are generated from corresponding high-level elements (i.e., the *Attribute* `dueDate` and the *ResourceAttribute* `dueDate`). In other words, these elements trace back to corresponding elements. Additionally, as is specified in the CDM (Figure 4.19) there is a *compatible constraint* between the metaclass *FieldDeclaration* in the business MTC and the metaclass *FieldDeclaration* in the security MTC. This means that we can generate a correspondence relationship between the *FieldDeclaration* `dueDate` and the private *FieldDeclaration* `dueDate`. Although there is a *compatible constraint* between the metaclass *MethodDeclaration* in the business MTC and the metaclass *MethodDeclaration* in the security MTC, it is not possible to identify the correct match between the setter *MethodDeclaration* `setDueDate`, and the

getter *MethodDeclaration* `getDueDate` in the *Mjava* model and the annotated *MethodDeclaration* `writeDueDate`. Intuitively a correspondence relationship must be created between the setter *MethodDeclaration* `setDueDate` and the annotated *MethodDeclaration* `writeDueDate`. A correspondence relationship should not be created between the getter *MethodDeclaration* `getDueDate` and the annotated *MethodDeclaration* `writeDueDate` because we want allow the *Role* `User` to modify the `dueDate` value.

Consequently, more than the type information is required to identify the correct matches between the target elements.

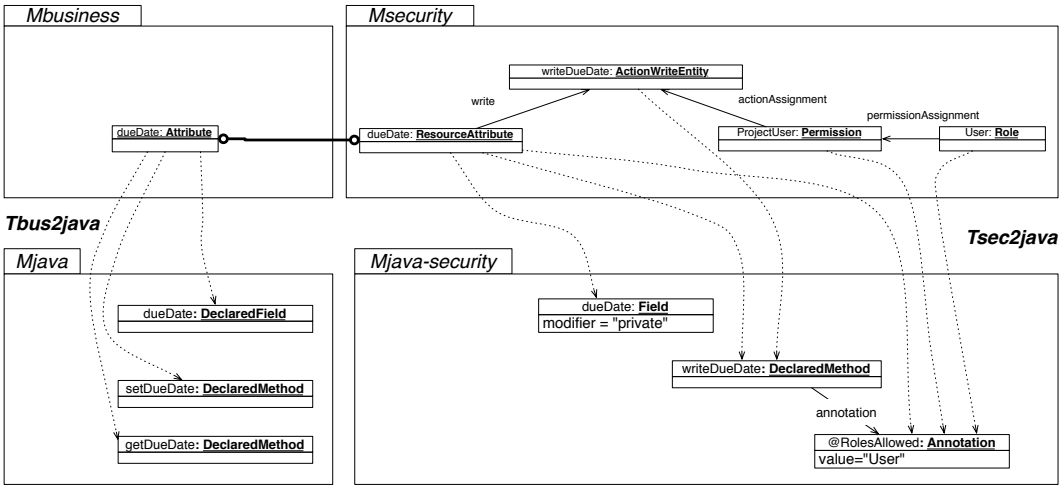


Figure 4.25: Insufficient Metaclass Information

4.7.1 Extending the tracing models

Keeping in mind that more information is needed to identify the correct match between the generated elements, we need additional information in the tracing models. This information is related to the name of the transformation rule that transformed the element and the name of the output variable. For instance, Listing 4.3 presents the rule that transforms an *Attribute* into a *FieldDeclaration*, the getter *MethodDeclaration*, and the setter *MethodDeclaration*. Additionally, Listing 4.4 shows the rule that transforms an `AttributeResource` into a private *FieldDeclaration* and an annotated `MethodDeclaration` when a `WriteAttributeAction` is protecting it.

Figure 4.26 illustrates the results of applying these transformations to our case study. In particular in this figure it is possible to see the tracing information tagged with the transformation rule name and the output variable name. For instance, the tracing link between the *Attribute* `dueDate` and the *MethodDeclaration* `setDueDate` is tagged with the rule name `Attribute2FieldDeclaration` and the output variable `setter`.

```
rule Attribute2FieldDeclaration {
  from
    attribute   : BUSINESS!Attribute
  to
    field     : JAVA!FieldDeclaration
    (
      name    <-  attribute.name
    ),
    getter      : JAVA!MethodDeclaration
    (
      name    <-  'get' + attribute.name
    ),
    setter      : JAVA!MethodDeclaration
    (
      name    <-  'set' + attribute.name
    )
}
```

Listing 4.3: Attribute to Declared Field Transformation

```
rule ResourceAttribute2FieldDeclaration {
  from
    attribute   : SEC!ResourceAttribute (
      not attribute.write.oclIsUndefined()
    )
  to
    field     : JAVA!FieldDeclaration
    (
      name    <-  attribute.name,
      modifier  <-  'private'
    ),
    writemethod     : JAVA!MethodDeclaration(
      name    <-  'write' + attribute.name,
      annotations <-  Sequence{annotation}
    ),
    annotation    : JAVA!Annotation(
      type    <-  'RolesAllowed',
      value   <-  attribute.actions->collect(e |
        e.permission->collect(f | f.role)
        )->flatten()
    )
}
```

Listing 4.4: Correspondence Model Transformation (CMT)

We modify the Tracing metamodel presented in Section 4.5.1 to store the information about the rule and the output variable. Additionally, the extension to the ATL-VM automatically adds this information to the tracing links.

4.7.2 Extending the Correspondence Derivation Model

Having in mind to use the tags in the tracing links, the CDMM is adapted by adding the name of the rules and the output variables to each side of a constraint link. Each *DerivationElement* is enriched with a the name of the rules and its output variables that are potentially compatible. This allows the *MTC developer* to specify the additional conditions to each constraint of the Correspondence Derivation Model in order to generate the correct matches when several elements conform to the same metaclass and share the same

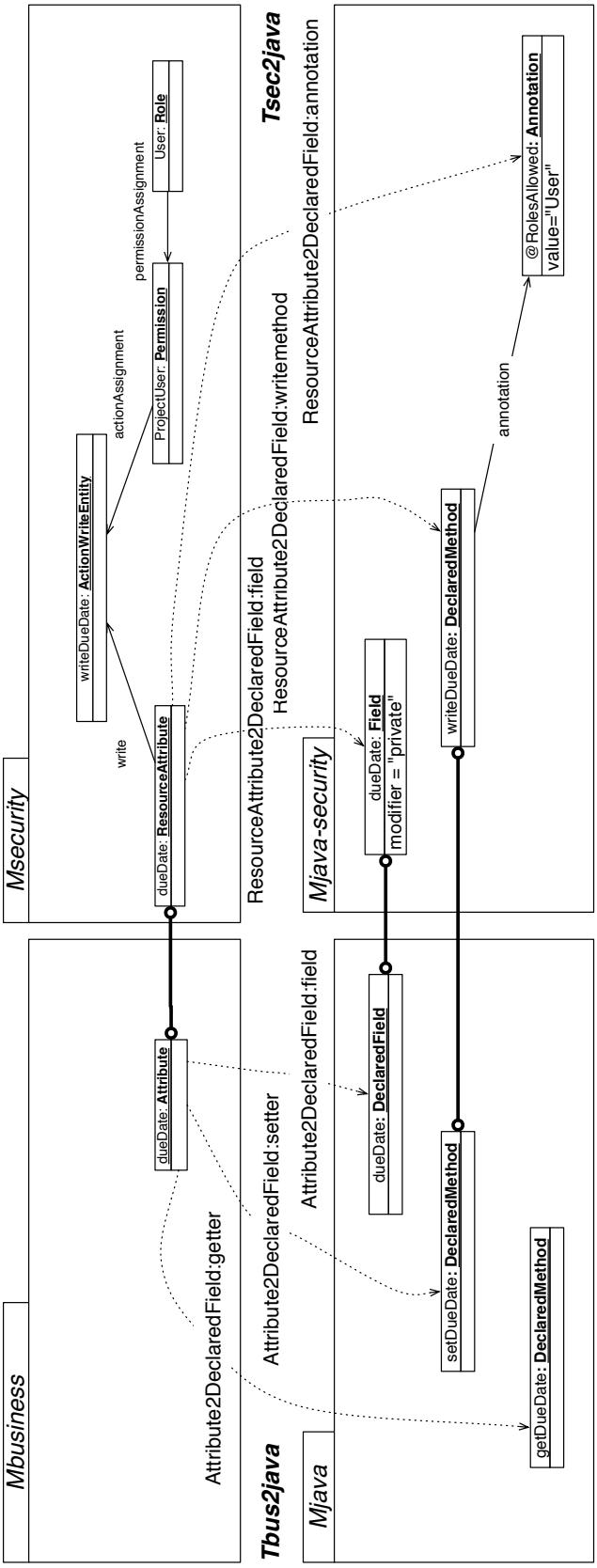


Figure 4.26: The extended information

source. Figure 4.27 shows the extended CDM for the presented example. The figure presents the extra condition as an annotation to the *MethodDeclaration*. This condition is:

$$
\begin{aligned}
(left.name = &\text{“}Attribute2FieldDeclaration : setter\text{”} \\
&and \\
right.name = &\text{“}ResourceAttribute2FieldDeclaration.writemethod\text{”}) \\
&or \\
(left.name = &\text{“}Attribute2FieldDeclaration : getter\text{”} \\
&and \\
right.name = &\text{“}ResourceAttribute2FieldDeclaration.readmethod\text{”})
\end{aligned}
$$

This condition requires that the *MethodDeclaration* in the $Mjava$ model must be generated by the *rule* `Attribute2FieldDeclaration` and the *output variable* `setter`[7] and the *MethodDeclaration* in the $Mjava - security$ model must be generated by the *rule* `ResourceAttribute2FieldDeclaration` and the *output variable* `writemethod`. If a couple of *MethodDeclarations* fulfill this condition and if these are generated from a couple of corresponding elements, then a correspondence relationship is created between them. Additionally, the condition is used to create correspondences between the getter methods and the read methods.

Listing 4.5 shows a fragment of the generated CMT for the extended Correspondence Model. The condition is presented in lines 10 to 14.

It is important to notice that there is a situation that makes it impossible to differentiate between two target elements. This situation occurs when both elements share the same source, conform to the same metaclass, and are tagged with the same rule name/output variable. This situation is originated when the tracing information is not correctly generated and does not allow the differentiation of all low-level elements. When this happens, the transformation used in Section 4.5.3 to compose the tracing models will alert the *MTC developer* about the problem. Two possible solutions are verifying if the required information is lost in the trace composition or extending the information stored in the tracing relationships.

4.8 Summary

In this chapter, the key element of our proposal was presented. This element is the automatic derivation mechanism that is responsible for generating correspondence relationships between the elements of the low-level models. This

[7]The name of a trace link is the name of the rule plus the name of the output variable.

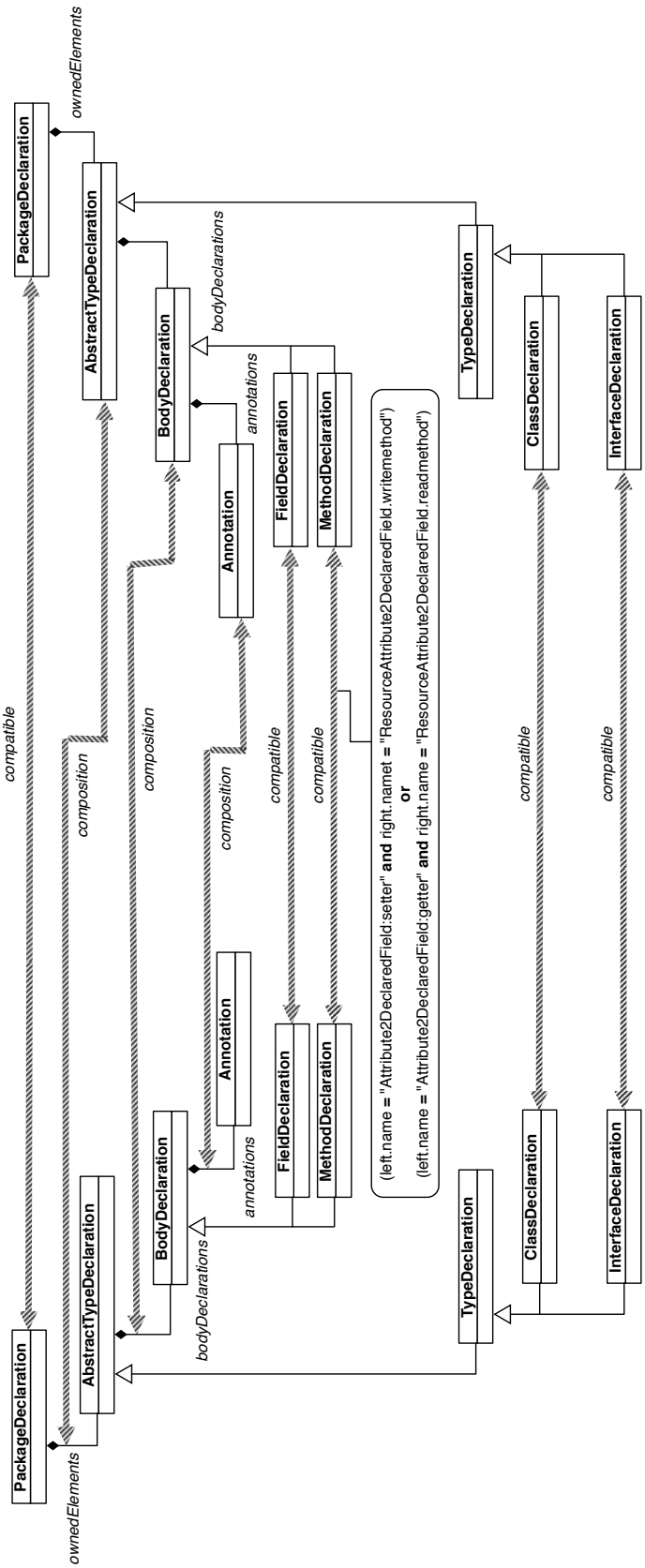


Figure 4.27: Extended Correspondence Derivation Model

```
1 rule EnhanceLink_MethodDeclaration_MethodDeclaration {
2   from
3     leftElement : LEFTOUT!MethodDeclaration,
4     rightElement : RIGHTOUT!MethodDeclaration,
5       inlink : CORRESPONDENCE!CompositionLink(
6         (inlink.left.getTargets('leftTrace')->flatten()->select(e |
7           e = leftElement)->notEmpty()) and
8         (inlink.right.getTargets('rightTrace')->flatten()->select(e |
9           e = rightElement)->notEmpty()) and (
10            (left.name = "Attribute2FieldDeclaration:setter" and
11            right.name = "ResourceAttribute2FieldDeclaration.writemethod")
12            or
13            (left.name = "Attribute2FieldDeclaration:getter" and
14            right.name = "ResourceAttribute2FieldDeclaration.readmethod")
15          )
16      )
17  to
18    outlink : CORRESPONDENCE!EnhanceLink (
19      model <- thisModule.matchModel,
20      left <- leftEnd,
21      right <- rightEnd
22    ),
23    leftEnd : CORRESPONDENCE!LeftElement (ref <- leftElement),
24    rightEnd : CORRESPONDENCE!RightElement (ref <- rightElement)
25  }
26 }
```

Listing 4.5: Correspondence Model Transformation (CMT)

derivation mechanism uses tracing information in order to identify elements that are generated from a couple of high-level corresponding elements.

The tracing information is generated in the form of tracing models. When the transformation rules are executed, the tracing models are generated automatically. These models relate each source element with their target elements. Additionally, each tracing link is tagged with the name of the rule and the output variable.

Furthermore, tracing information is not enough to identify the correct corresponding elements in the lowest level models. For this reason, the *MTC developer* needs to restrict the potential correspondence relationships between the low-level elements. The Correspondence Derivation Model defines constraint links between the target metaclasses. In these constraint links, it is specified if the derivation of a correspondence relationship is allowed or not. The CDM is processed by a High-Order Transformation that analyzes the constraint links and produces the Correspondence Model Transformation. The CMT is responsible for analyzing the models (i.e., high-level models, high-level correspondence model, tracing models, and low-level models) and producing the low-level correspondence model.

In the next chapter we elaborate on the Correspondence Model semantics and how the different correspondence relationships are resolved at the lowest-level of abstraction.

Chapter 5
Correspondence Relationships Resolution

5.1 Introduction

Once the existing *Model Transformation Chain* (MTC) and the new concern specific MTC produce the low-level models and the correspondence derivation mechanism produces the low-level correspondence model between these models, we need to *resolve* these relationships. For us the term *resolve* means to interpret the correspondence relationships and to process them in order to produce the application. We support three types of correspondence relationship resolutions: 1) *composition resolution* when the low-level models are complementary parts of the application that need to be integrated in order to produce the final application, 2) *checking resolution* when each model stands on its own and it is recommendable to check that they are consistent, and 3) *mapping resolution* when the responsibility of the integration is given to a platform specific code based mechanism and the correspondence relationships are translated into the composition specification of the used platform specific mechanism.

In this chapter we present how the *application modeler* identifies corresponding elements between the high-level models by using different types of relationships that allow him to express the type of resolution that must be performed to obtain the final application. The different types of relationships are explained in this chapter as well. Finally, we explain how we resolve each type of relationship using our three resolution mechanisms (i.e., composition, checking or mapping).

Figure 5.1 outlines the structure of this chapter based on the schema of our approach. In this chapter we present the semantics of a set of correspondence relationships used in this research. This set is specified in a Correspondence Metamodel that is presented in Section 5.2. This metamodel is used to define correspondence models that relate the high-level models. At a high-level of abstraction, the correspondence relationships are manually defined between

the high-level models by using simple types of relationship. In Section 5.3 we present how a high-level correspondence model is created using an example. Next, the relationships are propagated through the transformation steps by the correspondence derivation mechanism presented in Chapter 4. This correspondence derivation mechanism adds semantics to the existing correspondence relationships by using the extended metaclasses defined in the extende Correspondence Metamodel that is presented in Section 5.4. The types of relationships defined in this metamodel are used to specify correspondence relationships between low-level models. In Section 5.5 we present an example of a derived correspondence model between the lowest-level models that were generated by the MTCs. Additionally, when the relationships that connect the lowest-level models are obtained, we need to resolve them as explained before. Finally, in Section 5.6 we present the resolution mechanism we have implemented in order to obtain a complete model of the required application.

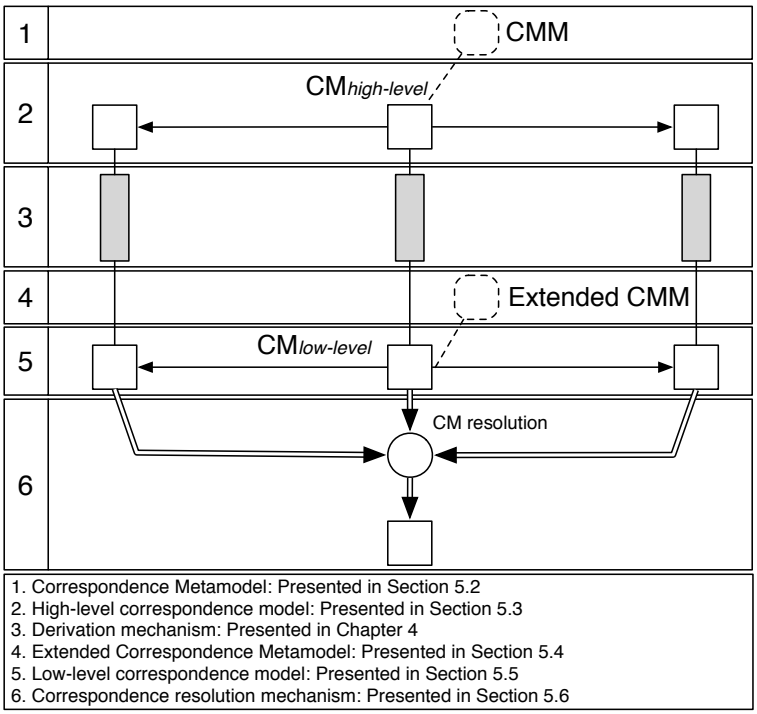


Figure 5.1: Chapter structure

5.2 Correspondence Metamodel

As presented in Section 2.4.2, a *Correspondence Model* (CM) represents relationships between elements of different models. These relationships must identify corresponding elements and the type of they will represent what kind of

resolution operation the modeler wants to perform between the corresponding elements. The different types of relationships are defined in a *Correspondence Metamodel* (CMM).

Figure 5.2 presents the Correspondence Metamodel that was defined in our research and was based on the ideas of the AMW Metamodel [DFBV06] presented in Section 2.4.2. The main elements of the Correspondence Metamodel are: 1) the *CorrespondenceModel* that represents the root element of the model, 2) the *CorrespondenceElement* that represents a corresponding element in one of the models, and 3) the *CorrespondenceLink* that represents a correspondence relationship between two corresponding elements in two different models.

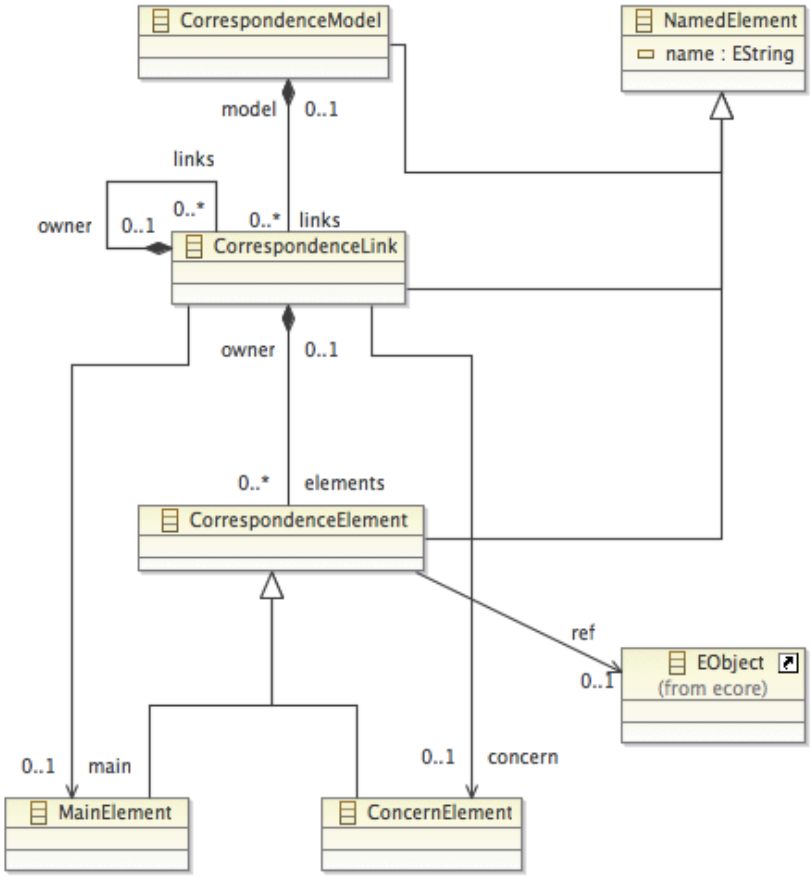


Figure 5.2: Correspondence Metamodel

The *CorrespondenceElement* is an extensible concept that has an *external* reference[1] to an EMF *EObject*. Having a reference to an EObject allows us to point to the actual EMF elements in the related models. This metaclass

[1] A *external reference* is a reference to a different model or metamodel

is extended into a *MainElement*, which represents an element in the existing model, and also into a *ConcernElement*, which represents an element in the new model.

A *CorrespondenceLink* relates two *CorrespondenceElements*, representing a correspondence relationship between two elements that belong to two different models. A *CorrespondenceLink* is a high-level relationship that needs to be extended in order to specify a concrete composition semantics.

5.2.1 Constraining relationships

The Correspondence metamodel allows us to connect any element in an EMF based model with any element in another EMF based model. This means that the *application modeler* will have a large amount of potential pairs of elements. In this large set of elements the *application modeler* will need to identify the correct matches. Having a huge amount of potential pairs can cause the modeler to make mistakes by specifying *false positives* and also to omit to specify some required pairs. In order to facilitate the creation of the CM, the *MTC developer* can extend this metamodel to constrain the possible correspondences to be defined by the *application modeler*.

The extension of the Correspondence Metamodel requires that the *MTC developer* performs three tasks:

1. Extending the metaclass *CorrespondenceElement* or its submetaclasses in order to constrain them to have an external reference to an element that conforms to a specific metaclass in the *left metamodel* instead of an *EObject*,

2. Extending the metaclass *CorrespondenceElement* or its submetaclasses in order to constrain them to have an external reference to an element that conforms to a specific metaclass in the *right metamodel* instead of an *EObject*, and

3. Extending the metaclass *CorrespondenceLink* in order to assure that it is only possible to define correspondence links between a pair of already constrained *CorrespondenceElements*.

Figure 5.3 shows an example of a possible extension for creating correspondence relationships between the Business and Security metamodels. In order to constrain the CM that is going to be created between a Business Model and a Security Model, a constrained link is defined by: 1) extending the metaclass *MainElement* into the metaclass *CorrespondenceBusinessEntity* that has an external reference to the metaclass *BusinessEntity* in the Business metamodel, and 2) extending the metaclass *ConcernElement* into the metaclass *CorrespondenceResourceEntity* that has an external reference

to the metaclass *ResourceEntity* in the Security Metamodel, and 3) extending the metaclass *CorrespondenceLink* into the metaclass *CorrespondenceLinkBusinessEntityToResourceEntity* that relates the metaclasses *CorrespondenceBusinessEntity* and *CorrespondenceResourceEntity*. Extending the metamodel will reduce the amount of possible pairs and will improve the usability of our solution.

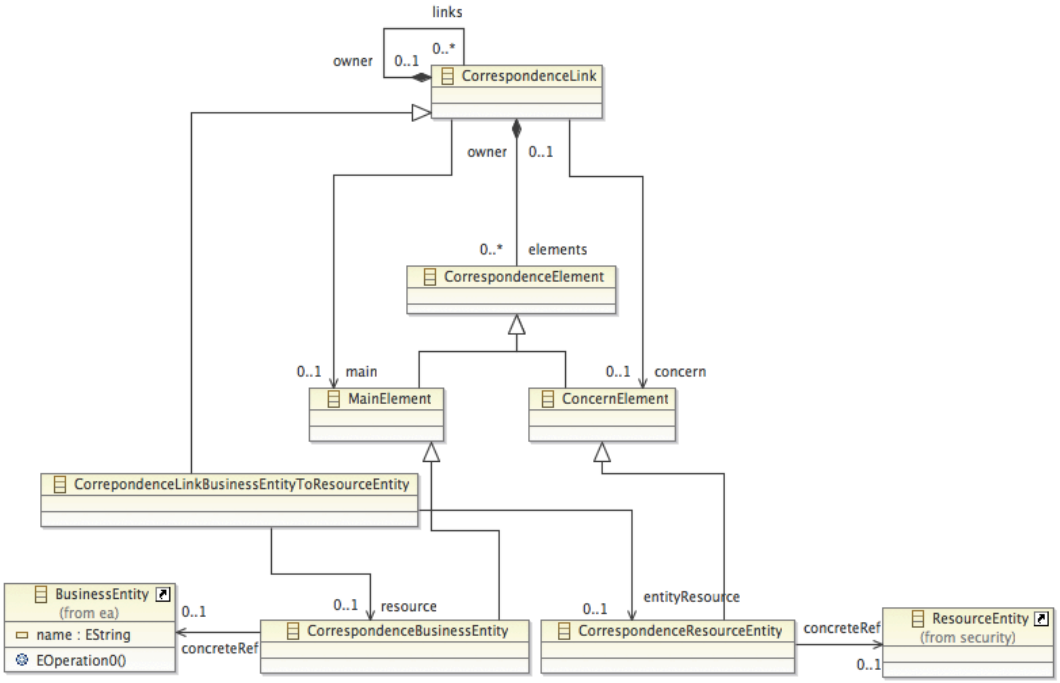


Figure 5.3: Correspondence Metamodel

The *MTC developer* must adapt the *Correspondence Modeler* presented in Section 6.4.1 in order to add constraints to the possible relationships that can be defined. If the *MTC developer* performs the adaptation to the *Correspondence Modeler*, then the tool will become metamodel specific for the pair of related metamodels.

5.3 High-level correspondences

Using the relationships defined in the correspondence metamodel, it is possible to specify correspondences between two models. At a high-level of abstraction, it is possible to define the CM using three strategies: *intensional*, *extensional*, or the combination of both [RJV09].

The first strategy is based on an *intensional* specification of the correspondence relationships. In this strategy, intensional formulas are used to infer the corresponding elements in both models. For instance, an intensional formula

could state that if an element that conforms to the metaclass *BusinessEntity* and an element that conforms to the metaclass *ResourceEntity* have the same name, a correspondence relationship between them should be created. There are several works that use various techniques and heuristics to automatically obtain such correspondences [GJCB09].

The second strategy is based on an *extensional* specification of the correspondence relationships. This means that every pair of corresponding elements is explicitly specified by the modeler. The extensional strategy tackles the problem of matching corresponding elements that requires human intervention to identify them. In our case we chose this strategy in order to focus on the derivation process and not on the definition of the high-level strategies.

In the third strategy, a combination of the previous two is used. Some corresponding elements are identified by intensional formulas, and the most complex ones are identified manually by a domain expert.

5.3.1 High-level Correspondence Metamodel Extension

The *application modeler* must identify the corresponding elements in the high-level models. Additionally, he needs to express the type of operation that he wants to perform with that pair of corresponding elements. At a high-level of abstraction, how these relationships are resolved is not completely defined and needs to be translated to the lowest-level of abstraction where the resolution mechanism is able to resolve the relationships. In our research, we propose three types of high-level correspondences that will represent the kind of resolution that will be performed at the low-level of abstraction. These types of relationships are:

CompositionLink: this relationship represents the intent of the *application modeler* to *integrate* the related elements. The composition of the related elements will be performed at the lowest level where it is clear how to integrate the models.

CheckLink: this relationship represents the intent of the *application modeler* to *check* the related elements. The checking of the related elements will be performed at the lowest level where every element is complete.

MapLink: this relationship represents the intent of the *application modeler* to *map* the related elements. The actual mapping of the related elements will be performed at the lowest level where the elements are totally generated.

We will define the refinement of each type links towards the lower levels later in this chapter. Though, first we will reintroduce our case study.

Example: Business and Security High-level Models

As presented in Section 4.3.2, we align the two high-level models using a CM that conforms to the extended Correspondence Metamodel presented in Section 5.3.1. In our example, the *application modeler* aims to integrate the Business Model and the Security Model presented in the previous chapter. He knows that these two models are complementary and that they need to be composed in order to produce a secure application. In Figure 5.4 the *Business Model* ($M_{business}$) contains the *BusinessEntity* `Project` and the *Security Model* ($M_{security}$) contains the *ResourceEntity* `Project` that needs to be protected. In this example the $M_{security}$ needs to be composed with the $M_{business}$ protecting the business resources. In this figure, the correspondence relationships between both models $CM_{high-level}$ are represented by black lines marked with *ComposeLink*. This is what the *application modeler* aims to do with the elements that he relates. In this case all of the corresponding elements need to be composed. The $CM_{high-level}$ contains a correspondence relationship with links to the *BusinessEntity* `Project` and the *ResourceEntity* `Project`. This relationship means that the *ResourceEntity* `Project` will be composed with the *BusinessEntity* `Project`. The composition of these two elements will cause the *BusinessEntity* to be protected. Moreover, the *Attribute* `dueDate` is related in the CM to the *ResourceAttribute* `dueDate`. The modeler manually creates these correspondence links because it is assumed that he knows which elements are corresponding and he understands the meaning of the relationships between the elements.

5.3.2 High-level heterogeneous composition

As presented in Section 2.4.2, if the models are expressed in different DSMLs, it is necessary to perform a *heterogeneous composition* (e.g., composition of a business entity from a business DSML and a secured resource from the security DSML). The use of heterogeneous DSMLs inherently increases the complexity of integrating multiple models.

Although it is possible to define the compositional semantics for a pair of high-level metamodels, if an unanticipated metamodel is added to an existing MTC, a huge amount of work will be required to specify the compositional semantics for the existing metamodel and the new unanticipated metamodel. Therefore, a way of reusing an existing composition mechanism is required.

In order to reuse a low-level composition mechanism and to reduce the complexity of evolving an existing MTC, an appropriate choice is to transform each model into a common metamodel. Using this strategy allows us to avoid performing a heterogeneous composition at a high-level of abstraction. Moreover, if the common metamodel is a technological platform metamodel (e.g., Java), it is possible to take advantage of an existing composition mechanism

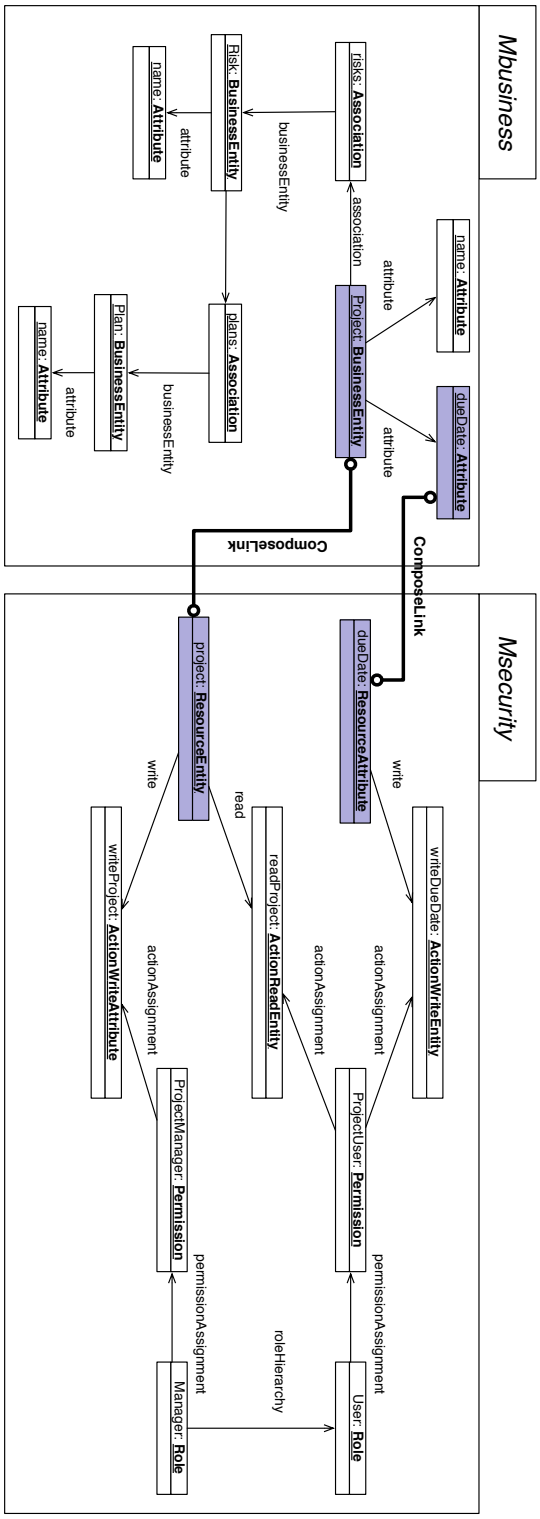


Figure 5.4: High-level Correspondence Model

for the platform. Usually, at the platform level there are several tested and commonly used composition mechanisms, such as AspectJ for Java.

If it is not possible to transform both models into a common metamodel, then the solution will be to transform both models into different platform based metamodels that have a composition mechanism defined (e.g., Java and SQL metamodels).

Therefore, to propagate the high-level correspondence relationships from the high-level to the low-level, we use the Correspondence Relationships Derivation mechanism presented in Chapter 4.

5.4 Extended Correspondence Metamodel

At the lowest-level of abstraction we need to define a concrete set of correspondence relationships that can be *resolved* by a reusable resolution mechanism, or an existing one, such as AOP.

In order to increase the level of reusability of the proposed correspondence relationships, we chose to have a set of correspondences that can be used to relate models that conform to any metamodel. Although this allows better reusability, using metamodel independent relationships restricts the expressiveness of the relationships.

We extend the *CompositionLink*, *CheckLink* and *MapLink* metaclasses in order to offer concrete compositional semantics between low-level models. The compositional semantics is implemented in a transformation rule that is written in ATL. This transformation rule allows us to resolve a set of correspondence relationships between two models and was inspired by the *UML Package merge* algorithm [DDZ08].

The concrete correspondence relationships that we propose allow us to perform the three types of resolution presented before. For the *CompositionLink* we propose to use composition operators that were inspired by several works in composition [SNEC06, BWH+08, Bri05]. These composition operators offer the required mechanisms to integrate two homogeneous models. Additionally, these composition operators are similar to the ones used in the composition of different views in an architecture description (AD) that were presented in Section 2.4.1. In the case of the extensions for the *CheckLink* relationships we offer a couple of checking operators that explicitly allow us to check consistency properties between two models. These checking operators are very simple and were implemented as proof of concept. More powerful consistency checking can be performed by using the *check-only* mode of *QVT Relations* [Obj09a]. Finally, we offer one extension for *MapLink* that can be used to generate code-level composition specification such as XML descriptors.

The types of relationships that we define in our resolution are explained next. Figure 5.5 shows the extended Correspondence Metamodel that contains

the new relationships. For each relationship we will state to which group of resolution type it belongs (i.e., Composition, Checking or Mapping), and if it allows us to relate homogeneous and/or heterogeneous models.

Table 5.1 presents a summary of the relationships provided. In this table each relationship is categorized by how it is resolved (i.e., Composing, Checking, or Mapping) and the kind of models that it can relate (i.e., heterogeneous or homogeneous models). Although it is possible to define more complex correspondence relationships such as those introduced in aspect-oriented programming [KLM+97], subject-oriented programming [HO93], and multidimensional separation of concerns [BLS03], they were not implemented in the context of this research. Instead, in this research we focus on the propagation of the relationships through the MTCs. The types of relationships defined in our work are:

	Compose	Check	Map	Homo-geneous	Hetero-geneous
Enhance	√			√	
Include	√			√	
Extend	√			√	
Override	√			√	
CheckSameName		√		√	√
CheckSameAttribute		√		√	√
MapsTo			√	√	√

Table 5.1: Correspondence Relationships Summary

Enhance: this is a *composition* relationship that enhances the element in the main model with the features of the element in the concern model. This relationship is only used between elements that conform to the same metaclass and belong to homogeneous models. This relationship is resolved by a model composition transformation that will add all the features from the concern model element into the main model element. For instance, a Java *Class* in the main model will be enhanced by the *Attributes*, *Methods* and *Annotations* of a Java *Class* in the concern model.

Includes: this is a *composition* relationship that connects two elements: the element in the concern model will be added as a feature of the element in the main model. This relationship is only used between homogeneous models which conform to different metaclasses. A containment relationship must exist between both metaclasses. This relationship is resolved by adding the element from the concern model into the main model element containment association. For instance, a Java *Class* in the main

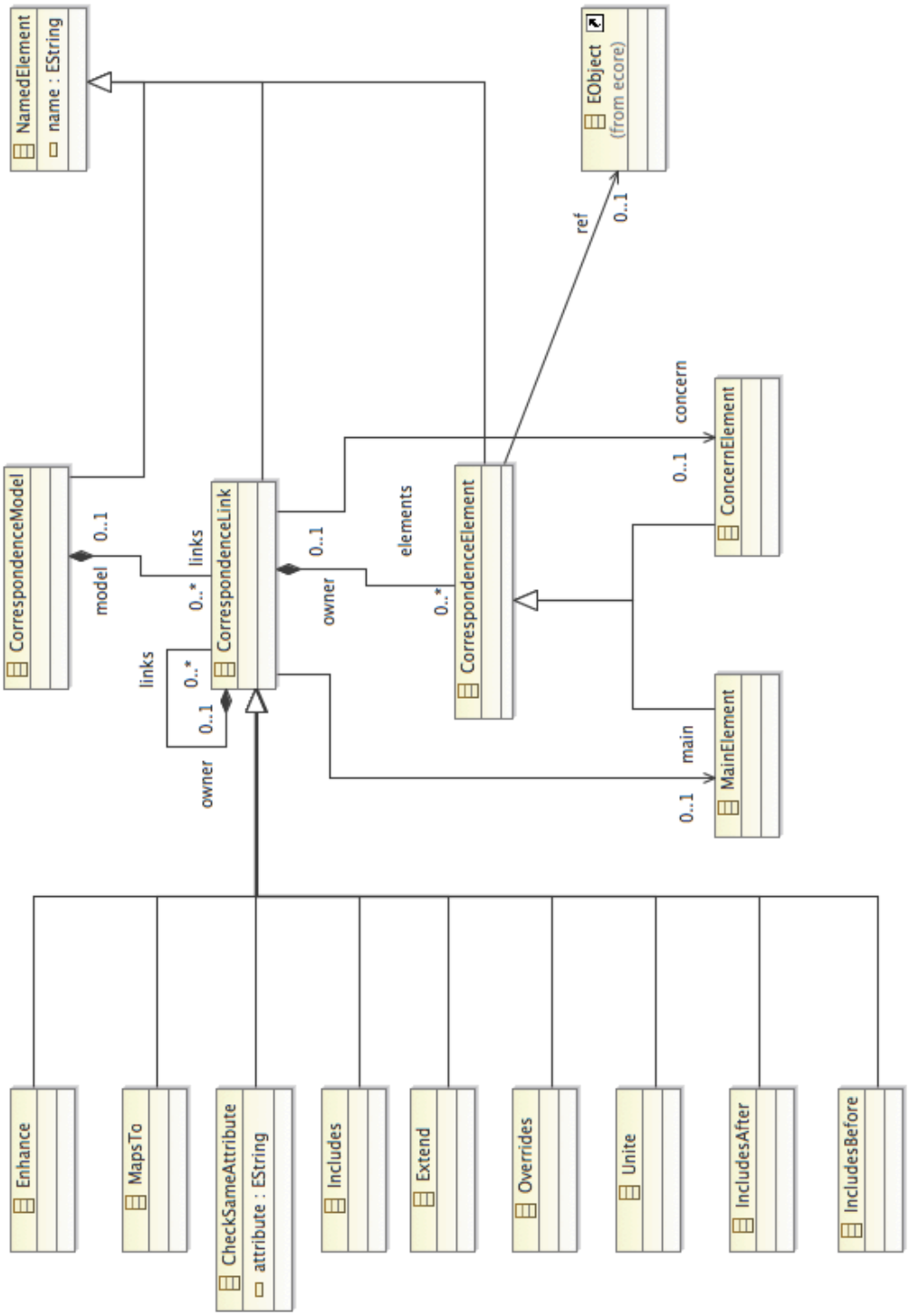


Figure 5.5: Extended Correspondence Metamodel

model is related with a Java *Attribute* in the concern model. When the composition is performed, the *Attribute* is added to the collection of *Attributes* of the Java *Class* in the main model .

Extend: this is a *composition* relationship that creates an *inheritance* relationship between the related elements that must conform to homogeneous metamodels. In order to create an inheritance relationship between the elements, the metamodels must support inheritance between their concepts (i.e., the metamodel of an object oriented language). This relation will connect the two lowest level classes. For instance, if two Java *Classes* are related, when the composition is performed, the *Class* in the main model will be the superclass of the *Class* in the concern model.

Override: this is a *composition* relationship that connects elements of the same type in homogeneous models. When the composition is performed, the element in the concern model will replace the element in the main model. For instance, if two Java *Methods* are connected, when the composition is performed, the *Method* in the concern model will replace the *Method* in the main model.

CheckSameName: this is a *checking* relationship that connects two elements that could belong to heterogeneous metamodels. When this relationship is resolved, a checking operation is performed between both elements by comparing if the elements have the same name. This relationship is used to check consistency between properties in the low-level models. For instance, a Java *Annotation* `@Table` in a Java model must have the same name of a *Table* that exists in an SQL model. This relationship can be used to detect if the *Annotation* and the table have the same name.

CheckSameAttribute: this is a *checking* relationship that connects two elements that could belong to heterogeneous metamodels. When this relationship is resolved, a checking operation is performed between both elements by comparing if the elements have an attribute (i.e., feature) with the same name and the same value. This relationship is used to check consistency between properties in the low-level models. The name of the *Attribute* to check must be specified in the relationship.

MapsTo: this is a *mapping* relationship that connects two elements. These elements belong to homogeneous or heterogeneous models. This relationship represents the same concept in different domains at a high-level of abstraction or two elements in different platforms at the lowest level of abstraction. At the lowest level of abstraction these two elements remain separated and will be connected by code level mechanisms. For

instance, a *Table* in a SQL schema and an *EJB Entity* in a Java application. This relationship could be transformed into a XML descriptor where the mapping of the elements is specified. Finally, these two elements are "composed" by the Application Server (e.g. JBoss, Glassfish) using the XML descriptor.

5.5 Low-level correspondences

The correspondence relationship derivation mechanism presented in Chapter 4 can generate a CM for both homogeneous and heterogeneous models. The generated CM will identify corresponding elements in low-level models that conform to the same metamodel or to different metamodels.

5.5.1 Correspondences between homogeneous models

When a couple of high-level models are processed by two MTCs, it is possible to generate two low-level models that conform to the same metamodel. Moreover, at the lowest level of abstraction, this metamodel could be based on the technological platform (e.g., Java metamodel); each of the two generated models is a complementary part of the final application.

On the one hand, the models can be orthogonal to each other without overlapping elements. For instance, each model could represent an application module, and the correspondence relationships could represent the dependencies between the modules. In this case the correspondence resolution mechanism must check for consistency problems between the generated models by comparing names or attributes of the related elements.

On the other hand, the generated models could have overlapping elements that need to be composed to generate the final application. In this case each one of the related elements can contain part of the information of the final element. The correspondence relationship represents how both parts should be integrated in order to obtain a complete version of the element. In this case the correspondence resolution mechanism must compose the models and produce a single model with all the information that was generated in the related models.

The main advantage of having both models conform to the same metamodel is that a single composition mechanism can be reused for any pair of concerns given that the concerns can be expressed using the common metamodel. Reusing a composition mechanism helps to evolve an MTC avoiding the need to specify new composition semantics for each new concern and to implement a new composition mechanism.

5.5.2 Correspondences between heterogeneous models

Depending on the application, at the lowest level of abstraction it is possible that the generated models conform to different metamodels. At this level, each metamodel is related to a technological platform (e.g., a Java metamodel or a JSP metamodel). In this situation each MTC transforms the high-level model into a different platform specific model. For instance, an MTC which takes as input a Business model and produces a Java Model while another MTC takes as input a Presentation model and produces a JSP model. These two models represent different parts of the required application, but they need to work together.

The correspondences generated between these two models should map elements that represent the same concept in the related platforms. For instance, a *Field* in the Java model could be mapped to a *TextField* in the JSP model. This will mean that the *TextField* represents the *FieldDeclaration* in the JSP platform and it will be responsible for presenting or capturing the value of the *FieldDeclaration* in the user interface.

At this level metamodels are based on the technological platform. Thus their semantics are clear and completely defined[2]. Similarly, how the involved platforms interact and are composed is clearly defined. For instance, how a Java application accesses a database or how a JSP calls methods in a Java application are clearly defined in the Application Server.

In summary, at the platform level, it is clear how each platform specific part of the application interacts with the other. Therefore, the ideal is to leave the composition of the different parts to the platform mechanisms. This means translating the correspondence relationships into a platform specific code-level composition specification. Additionally, the correspondence relationships can be used to check the consistency between the models.

5.6 Resolving correspondence relationships

When two models are generated from two complementary MTCs, and a Correspondence Model relates the models identifying corresponding elements, one has to *resolve* the correspondences. The term *resolution* in our work is defined as the interpretation of the correspondence relationships and processing them in order to produce the application. In our work we identify three possible scenarios that are *resolved* in different ways: 1) The first scenario is *composition*. In this situation the generated models are complementary parts of the application that have overlapping elements that need to be integrated in order to

[2]At a higher level of abstraction the semantics of the metamodels are specified in the transformations. Thus it is necessary to look into the transformations for fully understanding the semantics of each metamodel concept.

produce the final application. 2) The second scenario is *checking*. In this situation, the models represent different views, parts, subsystems or modules of the final application. These models can be translated directly into code without any composition operation. However, it is advisable to check that the models are consistent. We *resolve* the correspondences by performing a property check between the corresponding elements. 3) The third scenario is *mapping*. In this scenario, each model represents a model of the system that needs to be integrated. The responsibility of this integration is given to a platform specific code based mechanism. Therefore, the correspondence relationships should be translated into the composition specification of the platform specific mechanism used (e.g., XML descriptors in JEE).

5.6.1 Resolution Strategy: Composition

We offer a reusable composition mechanism that supports the previously mentioned relationships and is able to compose homogeneous models. The idea of the composition mechanism is to integrate two models that conform to the same metamodel and to produce a model that conforms to the same metamodel as well. Figure 5.6 shows a possible composition example. At the top part of the figure, the Business and the Security models are presented. These two models conform to the Java metamodel.The Correspondence Model relates the models identifying all the corresponding elements. Then in the middle of the figure the composition mechanism is represented as black arrows. The composition mechanism is a transformation rule that takes the two models and the CM as input and produces a single model as output. The generated model is presented in the bottom of the figure and is the full model of the application that conforms to the Java metamodel as well.

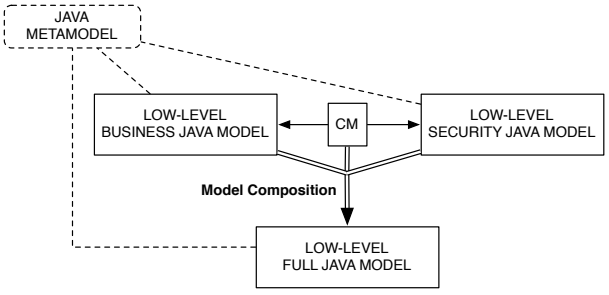


Figure 5.6: Homogeneous Model Composition

Composition strategy

The composition strategy that we use is based on classifying the elements of the input models into two groups: 1) the elements that are not related

by a correspondence relationship, and 2) the elements that are related by a correspondence relationship.

The second group is constituted by elements that are incomplete and need to be integrated in the final model. These elements are related by a correspondence relationship and the type of this relationship indicates how they need to be composed. In other words, the information of these elements is located in the two source models and needs to be integrated in the final model.

Figure 5.7 illustrates our composition strategy by means of the example that was presented in Section 4.2. In the figure, the elements that are not related by a correspondence relationship (i.e., in white) are copied directly to the target model without any modification. The elements that are related by a correspondence relationship (i.e., in color) are partial representations of the same concept and need to be integrated. For this reason they are composed and a single element is generated in the target model from each pair. This single element will have the information that was specified in the source pair. In the example we maintain the left side elements, but we include the relationships of the right side elements.

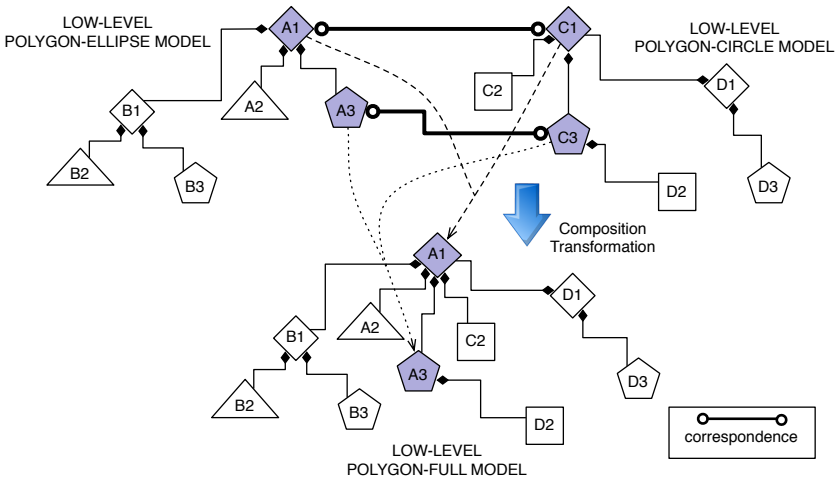


Figure 5.7: Homogeneous Model Composition

When a pair of corresponding elements is processed, the first step is to identify the type of correspondence relationship that connects them. Having identified this type, the composition transformation decides how to integrate both elements.

Relationships resolution

As presented before, the generated CM ($CM_{low-level}$) is an essential input for composing both low-level models in order to obtain a complete model of the application. The low-level CM model has the information of *what* will be

composed by identifying pairs of corresponding elements. Additionally, the type of the correspondence relationship is used to define *how* the two elements will be integrated. The composition transformation must resolve each type of composition relationship in order to produce an integrated model. The resolution of each type is presented next with the help of an example.

Enhance: this type of composition is resolved by adding the *features* of the concern element into the main element. Figure 5.8 shows the *Class* `Project` in the application low-level model (M_{java}) that is related with the *Class* `Project` in the security low-level model ($M_{sec-java}$). These two *Classes* are related by an *Enhance* relationship. When the composition is performed, the *Annotations* of the *Class* in the security low-level model are added to the *Class* in the application low-level model. A single *Class* `Project` is obtained with the features of the two related *Classes* (i.e., its *Attributes*, *Methods* and *Annotations*).

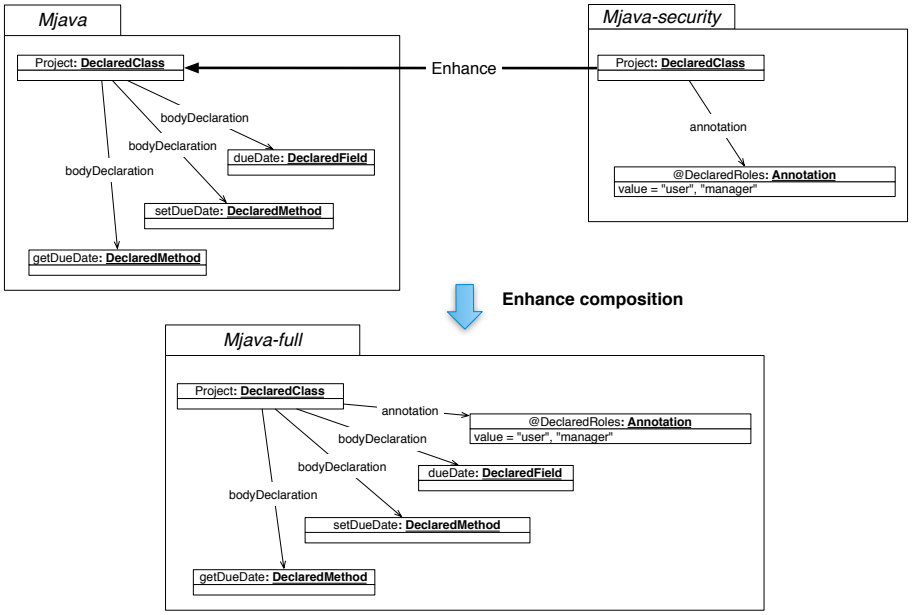


Figure 5.8: Enhance resolution

Include: this type of composition adds the element in the concern model as a feature of the element in the main model. Figure 5.9 shows the *Class* `Project` in the application low-level model (M_{java}) that is related with the *Attribute* `principal` in the security low-level model ($M_{sec-java}$). These two elements are related by an *Include* relationship. When the composition is performed, the *Attribute* is added to the collection of *Class* attributes. A single *Class* `Project` is obtained with the added *Attribute*.

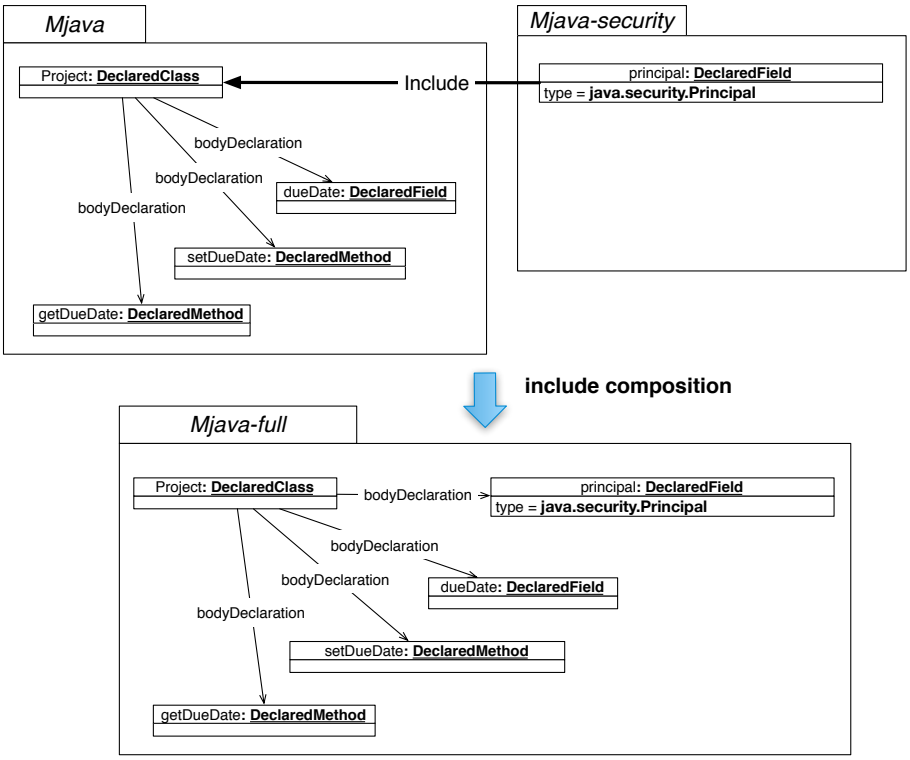


Figure 5.9: Include resolution

Extend: this type of composition relates two *Classes* with an inheritance relationship[3]. Figure 5.10 shows the *Class* `User` in the main low-level model that is related with the *Class* `Manager` in the concern low-level model. These two *Classes* are related by an *Extend* relationship. When the composition is performed, the two *Classes* are connected by a subtype relationship.

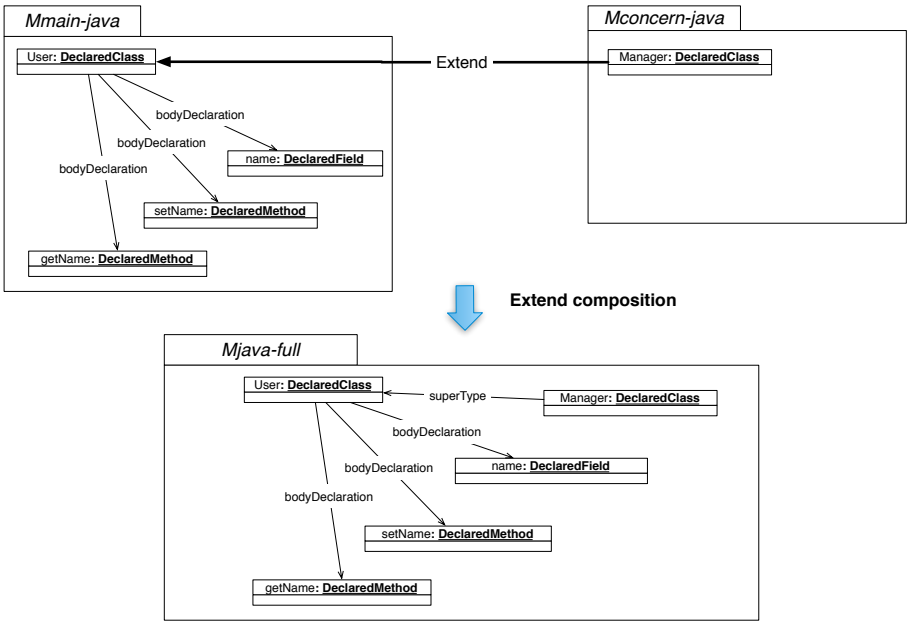


Figure 5.10: Extend resolution

Override: this type of composition relates two elements that conform to the same metaclass. When the composition is performed, the element in the concern model will replace the element in the main model. Figure 5.10 shows the *Attributes* **dueDate**. The *Attribute* in the main model has **public** access. The *Attribute* in the concern model has **private** access. When the composition is performed, the *Attribute* in the concern model replaces the *Attribute* in the main model. This means that the *Attribute* in the full model has *private* access.

Composition Transformation

In our approach we use the CDM to generate a composition transformation. The CDM is processed by a HOT and a composition transformation is produced. The HOT analyzes each compatibility relationship and produces two ATL matched rules. The first rule is responsible for copying the elements in

[3]Note that the extend relationship is only possible for metamodels that support a notion of object-oriented extensions.

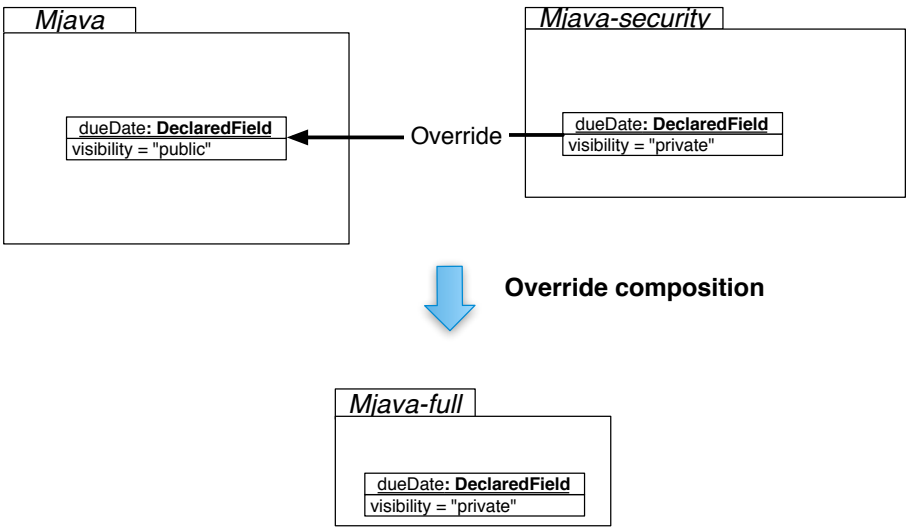


Figure 5.11: Override resolution

the main and concern models to the complete model without any modification. This rule specifically selects elements that are not related by a correspondence relationship. The second rule is responsible for composing a pair of corresponding elements that depend on the semantics of the target relationship in the CDM.

Listing 5.1 presents the three rules generated for the composition of a *PackageDeclaration*. In the CDM there is a *CompatibleLink* between the *PackageDeclaration* metaclasses and an *Enhance* composition relationship is selected as target. The first rule (i.e., `PackageDeclarationCopy`) is a copy rule that copies any package that is not related by a correspondence relationship. Lines 3-8 are the filter that allows us to select only the non-related elements. The second rule (i.e., `PackageDeclarationEnhanced`) is responsible for composing two corresponding *PackageDeclarations*. This rule performs a *union* of the metaclass structural features that have a cardinality greater than 1. If the cardinality is 1, the rule checks if a structural feature is defined in the main model. If it is defined, it copies its value to the final model. If it is not defined, then the value of the concern model is copied to the final model. In the example, `name` and *proxy* features have a cardinality of 1 and `comments`, `ownedElements` and `ownedPackages` greater than 1.

5.6.2 Resolution Strategy: Checking consistency

When the generated models are orthogonal and do not share overlapping elements, a potential use for correspondence relationships is to perform a consistency check between the related elements. For instance, they are used to verify if two related elements have the same name or if a given property has

```
1 rule PackageDeclarationCopy {
2  from
3   s : METAMODEL!"j2se5::PackageDeclaration" (
4   (thisModule.mainElements->includes(s) and
5   not thisModule.linkedMainElements->includes(s)) or
6   (thisModule.concernElements->includes(s) and
7   not thisModule.linkedConcernElements->includes(s))
8   )
9  to
10  t : METAMODEL!"j2se5::PackageDeclaration" (
11   name <- s.name,
12   proxy <- s.proxy,
13   comments <- s.comments,
14   ownedElements <- s.ownedElements,
15   ownedPackages <- s.ownedPackages
16  )
17 }
18
19 rule PackageDeclarationEnhanced {
20  from
21   s : METAMODEL!"j2se5::PackageDeclaration" (
22     thisModule.mainElements->includes(s) and
23     thisModule.linkedMainElements->includes(s))
24   using {
25   link  :   MATCHMM!AligmentLink = thisModule.getLink(s);
26   }
27   to t : METAMODEL!"j2se5::PackageDeclaration" (
28     name <- if not s.name.oclIsUndefined() then
29         s.name
30       else
31         link.right.ref.name
32       endif,
33     proxy <- if not s.proxy.oclIsUndefined() then
34         s.proxy
35       else
36         link.right.ref.proxy
37       endif,
38     comments <- s.comments->union(link.right.ref.comments),
39   ownedElements <- s.ownedElements->union(link.right.ref.ownedElements),
40     ownedPackages <- s.ownedPackages->union(link.right.ref.ownedPackages)
41   )
42 }
```

Listing 5.1: Attribute to Declared Field Transformation

the same value.

We implement a basic checking mechanism that helps the *application modeler* and the *MTC developer* to identify inconsistencies between the generated models. The potential inconsistencies in the generated models could be introduced as modeling errors in the high-level models, incorrect correspondences at high-level model, problems in the transformations or a poorly defined Correspondence Derivation Model.

Relationship resolution

The consistency checking mechanism that we implement only verifies basic properties between the models. The checking is performed by comparing only the elements that are related by a correspondence relationship. The elements

that are not related by a correspondence relationship are not checked.

For each pair of corresponding elements, a simple check is performed depending on the type of the relationship. Finally, a report is presented reflecting which corresponding elements fulfill the condition and which elements do not.

CheckSameName: this relationship connects two elements only when they have a feature called *name*. Additionally, the value of the feature *name* must be the same. For instance, a Java *Annotation* **@Table** in a Java model has a feature called *name* that must have the *name* of a *Table* that exists in an SQL model. This relationship can be used to detect if the *Annotation* and the *Table* have the same name. Figure 5.12 shows the *CheckSameName* relationship between the *Annotation* and the *Table*. In this case the result of the name check is *true* and no warnings are produced.

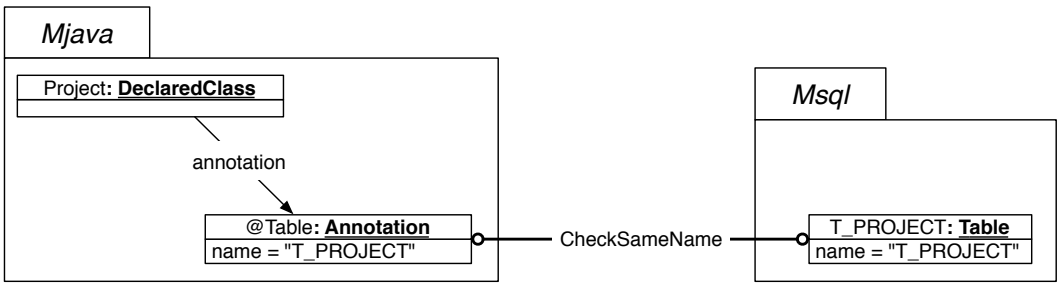


Figure 5.12: CheckSameName resolution

CheckSameAttribute: this relationship connects two elements only when they have the same specified attribute. This specified attribute must have the same name and the same value. This relationship is used to check consistency between properties in the low-level models. For instance, an MTC generates a Java model and the other MTC generates a JSP model. The JSP model must refer to an existing element in the Java model. Therefore, it is possible to use this type of checking to verify that the element in the JSP model has the same value as the element in the Java model. The name of the *Attribute* to be checked must be specified in the relationship.

5.6.3 Resolution Strategy: Mapping to code-level composition

In some situations when the two generated models conform to two different platform based metamodels (i.e., Java metamodel and SQL metamodel), the composition can be delegated to the platform specific composition mechanisms.

For instance, a JEE Application Server knows how to communicate with a database. Therefore, the responsibility of the correspondence relationships is to generate the code level specification of the composition. One example of this is generating the XML descriptors which configure the Java application to interact with the database.

Relationships resolution

MapsTo: this relationship connects two elements that represent the same concept in different domains at a high-level of abstraction or two elements in different platforms at the lowest level of abstraction. At the lowest level of abstraction, these two elements that remain separated will be connected by code level mechanisms. For example, a *Table* in an SQL schema and a *EJB Entity* in a Java application. This relationship could be transformed into an XML descriptor where the mapping of the elements is specified. Finally, these two elements are "composed" by the Application Server (e.g. JBoss, Glassfish) using the XML descriptor.

Figure 5.13 shows the correspondences used to generate an XML descriptor between the Java model and the SQL model. In this case, annotations are not used and the relations are created between the *ClassDeclarations* and also the *Tables*, and the *FieldDeclarations* and *Columns*. For example, there is a *MapTo* correspondence relationship between the *ClassDeclaration* `Project` and the *Table* `T_PROJECT`. Another *MapTo* correspondence relationship is derived between the *FieldDeclaration* `name` and the *Column* `PROJECT_NAME`.

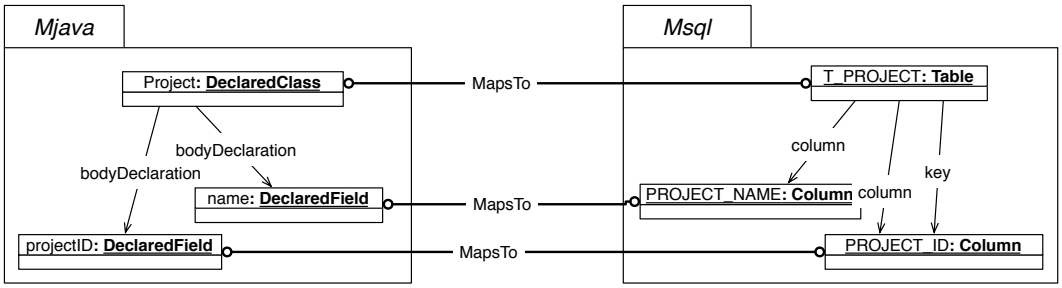


Figure 5.13: MapsTo resolution

Once the correspondence model is generated, it is used by a code generator that transforms each relationship into XML code. The generated XML code is presented in Listing 5.2. This XML illustrates how each *ClassDeclaration* is mapped to its corresponding *Table* and how every *FieldDeclaration* is mapped to its corresponding *Column*. Finally, when the application is executed, the two models are composed.

```
<entity-mappings>
  <entity class="Project"
      metadata-complete="true">
    <table name="T_PROJECT" />
      <attributes>
        <id name="id">
          <column name="PROJECT_ID"/>
        </id>
        <basic name="name">
          <column name="PROJECT_NAME"/>
        </basic>
        ...
      </attributes>
    </table>
  </entity>
  ...
</entity-mappings>
```

Listing 5.2: Transformation Rule ($T_{bus2sql}$)

5.7 Summary

In summary, this chapter presents the semantics of the correspondence relationships and how each type of correspondence can be resolved. The semantics of the correspondence relationship is defined in a Correspondence Metamodel. This metamodel can be extended in two different ways: 1) by adding constraints to the types of elements that can be related, and 2) by adding new types of correspondences. We show how we can perform these two types of extensions.

At the high-level of abstraction, we manually define the correspondence relationships that are used by our correspondence derivation mechanism in order to generate a low-level CM. The correspondence derivation mechanism adds semantics to the relationships by selecting from the different types of relationships defined in the Correspondence Metamodel. These low-level correspondences must be resolved in order to obtain the final application. The resolution of the low-level correspondences can be performed in three different ways: 1) composing the related models, 2) checking consistency between the models, and 3) transformed into code-level composition specifications, such as XML descriptors.

Chapter 6

Tool Support

6.1 Introduction

We have developed a set of proof-of-concept tools, called *MTC Framework Toolkit* to support the evolution of an existing MTC and the generation of applications using an evolved MTC. We present this set of tools keeping in mind the target users. The first group of tools helps the *MTC developer* to evolve an existing MTC. The second group supports the *application modeler* by generating an application using an evolved MTC.

On the one hand, the main goal of the toolkit targeted for the *MTC developer* is to support him in the evolution of an existing MTC. This means to assist the developer in the alignment of two MTCs and in the generation of the required transformation rules to be able to compose the generated models or to check the consistency of the generated models.

On the other hand, the main goals of the toolkit targeted for the *application modeler* is to help him: 1) to define correspondence relationships between the high-level models, and 2) to execute the different transformations that derive the correspondences and compose or check models.

In the first part of this chapter we will use the security case study used in the previous chapters to illustrate how our tools help each target user in performing his tasks. The remainder of this chapter is structured as follows: Section 6.2 describes the architecture of the *MTC Framework Toolkit* components. Section 6.3 explains all the tasks that the *MTC developer* must perform to evolve an existing MTC and the tools that we provide him to assist in this process. Section 6.4 presents the tasks that the *application modeler* must execute in order to produce an application using the evolved MTC. Next, we present an ATL-VM extension that allows us to automatically generate tracing models. Finally, Section 6.6 summarizes this chapter.

6.2 Architecture Overview

The *MTC Framework Toolkit* consists of six components. Three of them support the *MTC developer* tasks, two of them support the *application modeler*, and one is for infrastructure support. These components make use of two third-party components. These components are the Eclipse Modeling Framework (EMF)[SBPM09] and the Atlas Transformation Language (ATL)[JK06]. Figure 6.1 gives an overview of the components and how they are related to each other.

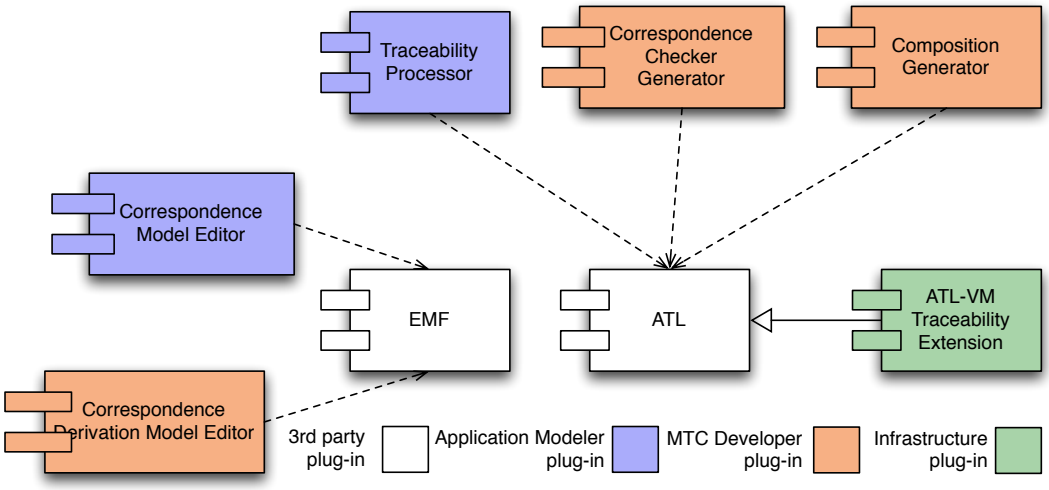


Figure 6.1: MTC Framework Toolkit architectural overview

The MTC Framework Toolkit Eclipse plug-ins add functionality to the Eclipse Integrated Development Environment to support the evolution of an MTC. On the one hand, the toolkit offers two model editors that allow the *MTC developer* and the *application modeler* to specify the Correspondence Derivation Model (CMD) and the Correspondence Model (CM) respectively. These two model editors are built using EMF as the modeling framework. On the other hand, we implement a set of tools that use the ATL framework to generate models, analyze models and generate new transformations. Additionally, we extend the ATL Virtual Machine (ATL-VM) in order to automatically generate tracing models.

Figure 6.2 shows how each plug-in supports our approach to evolve an existing MTC. In the following sections, each plug-in is explained using the case study of adding the security concern to an existing MTC. With the help of this case study, we illustrate the steps that the *MTC developer* and the *application modeler* must follow and the tools used in each step.

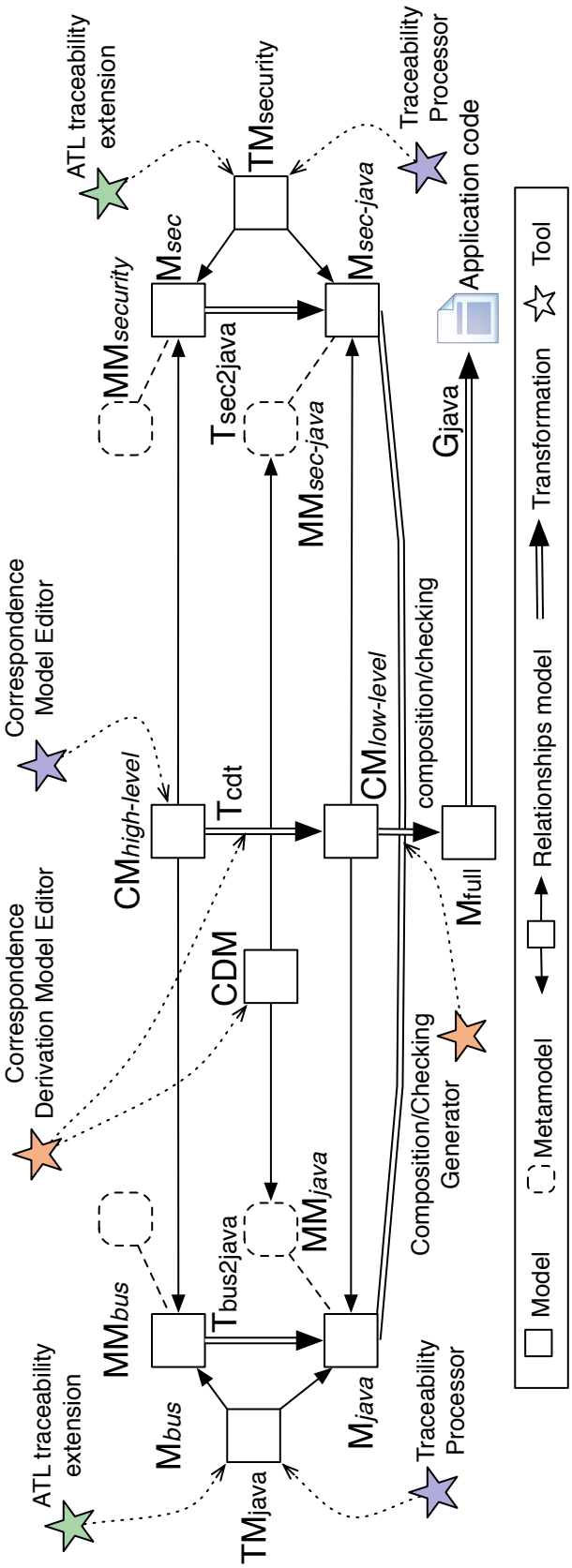


Figure 6.2: MTC Framework Toolkit support

6.3 MTC Developer Tasks

The first step that the *MTC developer* must perform is to implement a new concern-specific MTC (e.g., Security MTC). This new concern-specific MTC must transform a high-level model that conforms to a concern-specific metamodel and generate a low-level concern-specific model that conforms to the same low-level metamodel used by the existing MTC.

The second step that the *MTC developer* needs to perform is to use a tracing mechanism for the MTCs, such as the *ATL traceability extension* that we have provided or to manually modify the transformation rules to generate tracing models.

Next, the *MTC developer* needs to define the Correspondence Derivation Model (CDM) with the help of the *Correspondence Derivation Model Editor*. The CDM must contain constraints between the target metamodel specifying how the correspondences are propagated through the MTCs. Once the CDM is defined, it is necessary to generate the Correspondence Model Transformation (CMT). The *Correspondence Derivation Model Editor* allows the *MTC developer* to automatically generate an ATL transformation with the derivation constraints in it.

Finally, the *MTC developer* should generate both the composition transformation and the correspondence checker transformation. These two transformations are generated using the CMD defined in the last step. The *Composition Generator* and the *Correspondence Checker Generator* will process the CMD and generate the ATL transformations that compose the models or check the models respectively. The final result is the evolved MTC, ready to be used by the application modeler.

6.3.1 Correspondence Derivation Model Editor

Once the new concern-specific MTC is implemented and the tracing support has been added to both MTCs (i.e., existing and concern-specific MTC), the *MTC developer* should define how the correspondence relationships are propagated through the transformations in the CDM. Once the CDM is defined, the information in it is used to obtain the CMT. In order to perform these tasks, the *MTC developer* should use the *Correspondence Derivation Model Editor*.

The *Correspondence Derivation Model Editor* is the prime plug-in in our approach and helps the developer to visualize the target metamodels and the transformations in order to define derivation constraints between them. Additionally, the Correspondence Derivation Model Editor allows the *MTC developer* to automatically generate the CMT using the derivation constraints as input. Figure 6.3 shows a screenshot of the Correspondence Derivation Model Editor plug-in with an empty model.

This is an EMF based model editor that allows us to load the two target

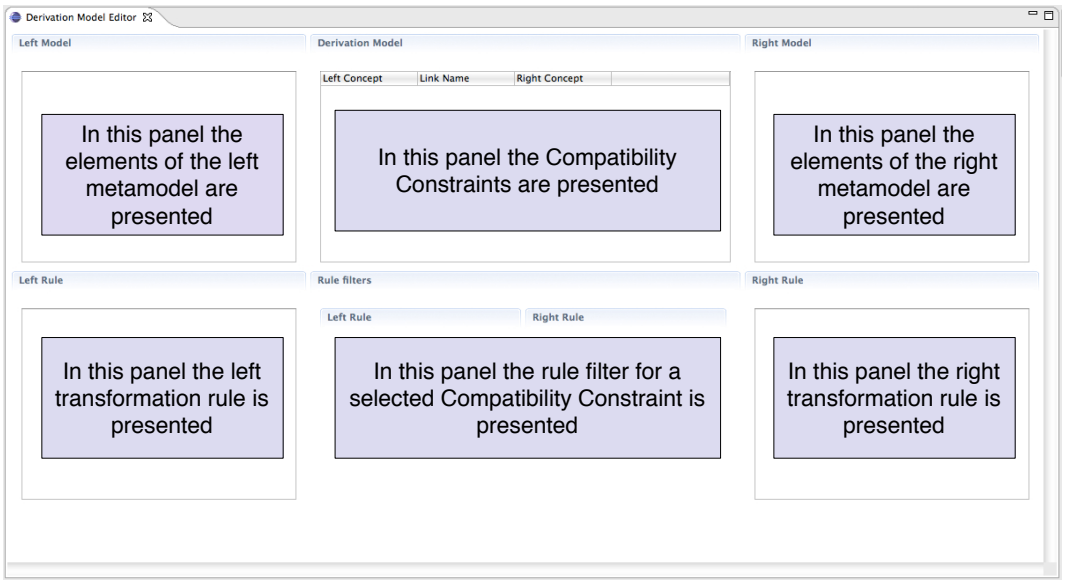


Figure 6.3: Correspondence Derivation Model Editor

metamodels as well as the two transformations. When the metamodels and the transformations for each MTC are loaded, the plug-in shows them in the left and right sections of the model editor. In the middle section, the model editor shows each constraint relationship defined in the CDM and the transformation filters applied to each constraint. Furthermore, the Correspondence Derivation Model Editor adds a set of action buttons to the eclipse toolbar that allow the *MTC developer* to load the metamodels, transformations or an existing CDM. Additionally, these buttons allow the *MTC developer* to create the multiple compatibility constraints defined in Section 4.6.2.

In the following subsections, we will use the security case study presented in Section 4.3 to illustrate the functionality of the Correspondence Derivation Model Editor and how it supports the *MTC developer*.

A. Create a new Correspondence Derivation Model

The first step is to create a new CDM for the existing MTC and the new concern-specific MTC. The Correspondence Derivation Model Editor plug-in will allow the *MTC developer* to create one empty CDM.

B. Load the target Metamodels

Once the empty model editor window is open, the next step is to load the target metamodels. A metamodel chooser dialog is provided by the plug-in to select the two target metamodels. Figure 6.4 shows the metamodel chooser dialog. This dialog only shows the metamodels that exist in the EMF metamodel

registry.

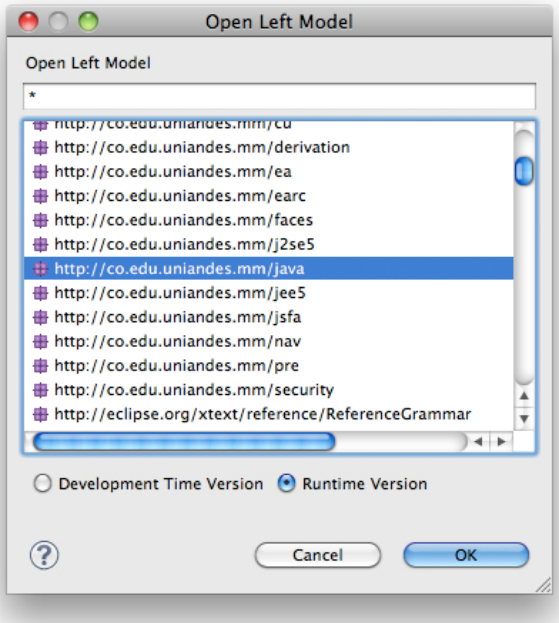


Figure 6.4: Metamodel chooser dialog

In our case study, the target metamodel for both MTCs is the Java Metamodel. Therefore, we load the Java metamodel in both sides. The metamodel chooser dialog shows all the metamodels that are listed in the EMF registry.

C. Compatibility Constraints creation

Once the target metamodels are selected, the *MTC developer* creates compatibility constraints which are the mechanisms used to define how the correspondence relationships will be propagated. This was previously explained in Section 4.6.

As presented in Section 4.6.2, there are four kinds of derivation constraints. For each kind the Correspondence Derivation Model Editor has a button to add a constraint in the CDM. A constraint is defined by selecting a pair of compatible metaclasses in the target models.

For instance, in our case study the metaclasses *MethodDeclaration* and *MethodDeclaration* are compatible. Hence, to define a compatibility constraint between this pair of metaclasses it is necessary to choose the *MethodDeclaration* metaclass in both sides of the model editor. When the metaclasses are selected, the type of desired constraint is selected, and a *CompatibleLink* is created between the selected metaclasses. In Figure 6.5 the target metamodels

are loaded and the *MethodDeclaration* and *MethodDeclaration* metaclasses are selected.

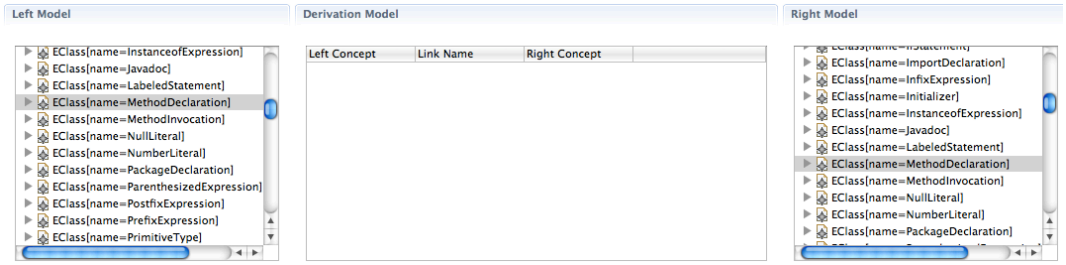


Figure 6.5: Creation of a Compatibility Constraint

Similarly, the different compatibility constraints are defined until the complete CDM is specified. Figure 6.6 presents the CDM that was shown in Figure 4.19 in Section 4.6.1.

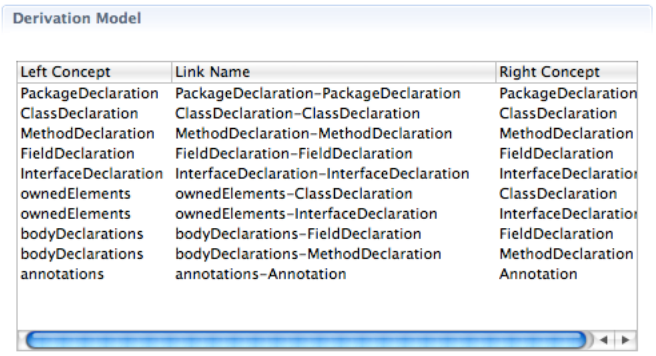


Figure 6.6: Correspondence Derivation Model

In Figure 6.6 each derivation constraint is shown with the name of the related metaclasses on each side and the type of the constraint in the middle. When a compatibility constraint is selected, the model editor is refreshed automatically by selecting the related metaclasses.

D. Filtering Derivation Constraints with Rule information

As presented in Section 4.7, when a transformation rule produces several elements that conform to the same metaclass, the metaclass information is not enough to define compatibility constraints. In this situation the *MTC developer* must add information about the transformation rule and the output variable. This information will be matched with the information added to the tracing models.

The *MTC developer* must add the left and right transformation rules using the Rule chooser dialog presented in Figure 6.7. This dialog only shows the transformation rules that exist in the Eclipse workspace.

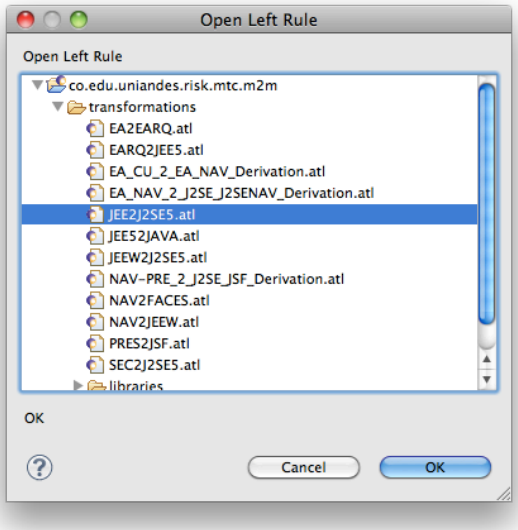


Figure 6.7: Transformation Rule chooser dialog

In our case study the *MTC developer* needs to choose the final transformations of each MTC. The ATL module for the final transformation of the Business to Java MTC is called `JEE2J2SE5.atl` and the ATL module for the transformation from Security to Java MTC is called `SEC2J2SE5.atl`. When both are added to the model, every time that an element is selected in the metamodel window, the list of rules is filtered by the metaclass type. This means that only the rules with an output of the chosen type are shown. For instance, Figure 6.8 shows a derivation constraint between the metaclasses *MethodDeclaration* and *MethodDeclaration*. In order to be able to restrict the application of this derivation constraint to only some elements, a rule filter was added. This particular rule filter is only applied when both elements conform to *MethodDeclaration* and were generated by the rules `Attribute2FieldDeclaration` in the left rule and `ResourceAttribute2WriteFieldDeclaration` in the right rule. Additionally, the name of the output variable in the left is `setter` and the output variable in the right is `writemethod`. The information of the rule filter is presented in the bottom-middle section of Figure 6.8.

This filter creates a correct correspondence between the field setter and the annotated method in order to write or modify a field, but does not create an incorrect correspondence between the field getter and the annotated method in order to read a field.

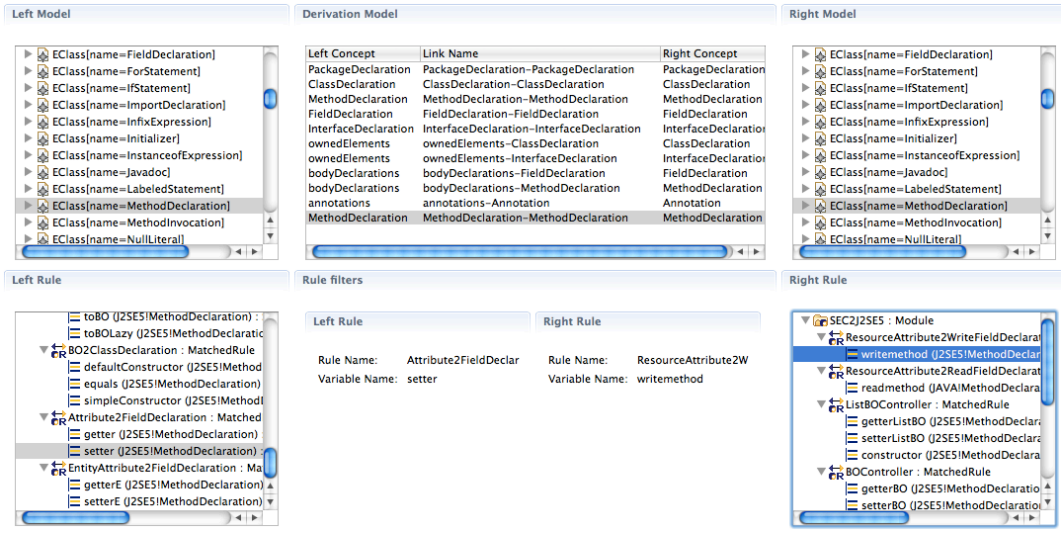


Figure 6.8: Correspondence Derivation Model

Using *rule name/output variable* filters, we extend the scope of the correspondence derivation mechanism by increasing the differentiation capabilities between the generated elements.

E. Generating the Correspondence Derivation Transformation

Once the CDM is complete, the developer must generate the Correspondence Derivation Transformation. As previously explained in Section 4.6.7, the CDM is defined at the metamodel level and cannot be used to directly generate the correspondence relationships. The CDM must be processed by a High-Order Transformation (HOT) and a transformation rule will be generated from it. This transformation will have the constraints specified in the CDM. Figure 6.9 shows the dialog that is opened when the *MTC developer* performs the generation. In this dialog the developer must choose the directory and file name of the Correspondence Derivation Transformation.

6.3.2 Composition Generator

The *Composition Generator* plug-in supports the *MTC developer* during the generation of a transformation rule that composes two models that conform to the same metamodel. This plug-in is implemented as an Eclipse view that allows the *MTC developer* to select the CDM and use the compatibility constraints to generate a transformation in order to compose to two models.

This composition generator is based on the compositional semantics presented in Section 5.6 where each pair of elements related by a correspondence relationship is composed. The plug-in processes the CDM and produces an

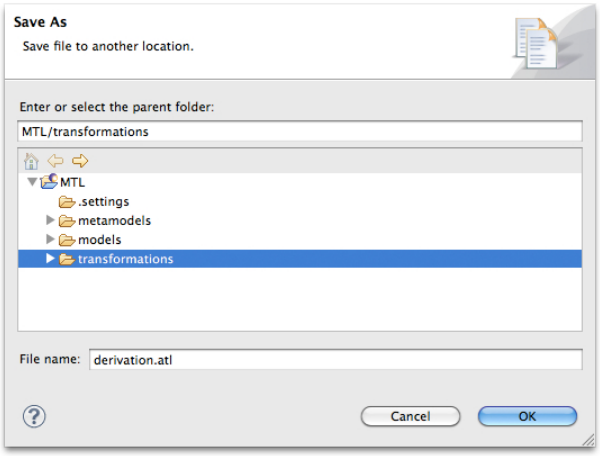


Figure 6.9: Correspondence Derivation Model

ATL transformation. For each compatibility relationship, the plug-in analyzes the output correspondence and produces a composition rule using the specified semantics. This generator is based on the use of an ATL High-Order transformation that takes the CDM and the metamodels related by it, and generates a composition transformation. For each compatibility link, the HOT produces a rule that allows the composition of two models. A generated rule was presented in Section 5.6.1.

6.3.3 Consistency Checker Generator

The *Consistency Checker Generator* is similar to the previous plug-in. This tool analyzes the CDM and identifies constraints that have as a target a *CheckLink*, for each of them, the tool creates an ATL rule that compares two elements in the target model. On the one hand, if the target link was a *CheckSameName*, the rule verifies that the structural feature `name` in both models has the same value. On the other hand, if the target link was a *CheckSameAttribute*, the rule verifies that the `given` structural feature in both models has the same value. Similarly to the Composition Generator, this plug-in uses a HOT to analyze the CDM.

6.4 Application modeler Tasks

In the case of the *application modeler*, the main goal of the MTC Framework Toolkit is to help him to define correspondence relationships between the high-level models. Having in mind this goal, an *application modeler* has to perform the following steps:

1. Defining a new application model or choosing an existing one, such as the existing Business Model presented in Section 3.2.1

2. Defining the new concern model, such as the Security model presented in Section 4.3.3.

3. Specifying the correspondences between the different models using the *Correspondence Model Editor* plug-in.

4. Executing the MTC transformations.

5. Checking the tracing models for ambiguities using the *Traceability Processor* plug-in.

6. Generating the low-level correspondence model by executing the CDT generated by the *MTC developer*.

7. Composing/Checking the generated models by using the composing/checking transformations generated by the *MTC developer*.

8. Generating the application code using the existing model-to-text transformation.

The final result of performing all these tasks is the desired application code.

6.4.1 Correspondence Model Editor

Once the new concern-specific MTC is added to the existing MTC by the *MTC developer* and all of the derivation and composition transformations are generated using the presented plug-ins, the *application modeler* uses the evolved MTC to generate applications.

A developer has two options: the first one is to use an existing high-level model (e.g., the existing Business Model) and to specify a new one for the new specific concern, or the second option is to specify the two models from scratch. In both situations, once both of the models are ready, the *application modeler* must identify the corresponding elements that exist between both high-level models. A correspondence relationship must be defined between each pair of corresponding elements by the *application modeler*. The developer is assisted by the *Correspondence Model Editor* plug-in in order to identify the corresponding elements and to define the high-level correspondence model.

In the following subsections, we will continue using the security case study presented in Section 4.3 to illustrate the functionality of the Correspondence Model Editor and how it supports the *application modeler*.

A. Creating a new Correspondence Model

The first step is to create a new CM by using the Correspondence Model Editor plug-in. This plug-in is implemented as an Eclipse view in order to allow the definition of the CM without closing other model editors.

The bottom of Figure 6.10 shows a screenshot of the Correspondence Model Editor view with an empty model. In the top part of Figure 6.10, the `Risk` business model is presented using a *Topcased*[1] generated model editor. As previously presented, it is possible to see the model (e.g., business model) and the CM at the same time, helping the *application modeler* to specify the CM.

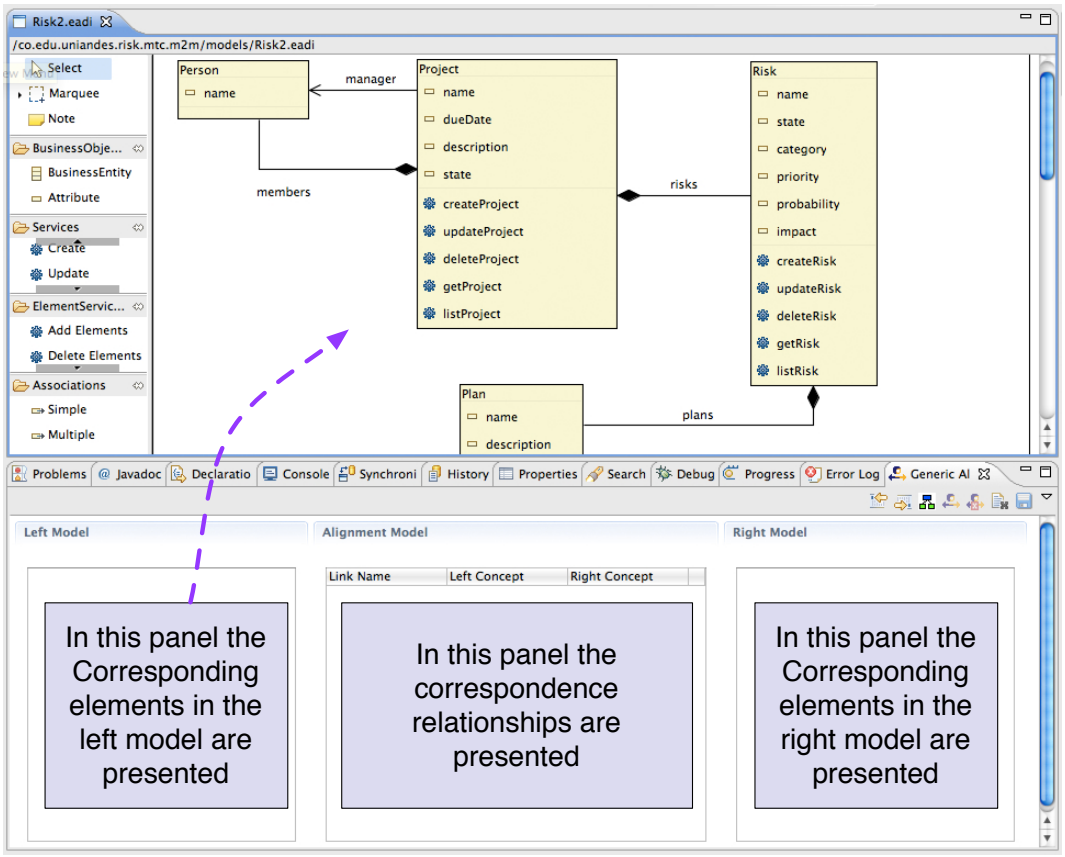


Figure 6.10: Correspondence Model Editor

The plug-in is divided into three sections: 1) the left section that shows the main model, 2) the middle section that presents the CM and its correspondence relationships, 3) the right section that shows the concern model. Additionally, in the top-right part of the view, there are action buttons to load the models, edit the CM and save it.

[1] http://www.topcased.org

B. Loading the corresponding Models

Once the empty model editor view is open, the first step is to load the models. The plug-in provides a model chooser dialog that allows the *application modeler* to select among the models located in the Eclipse workspace. Figure 6.11 shows the model chooser dialog.

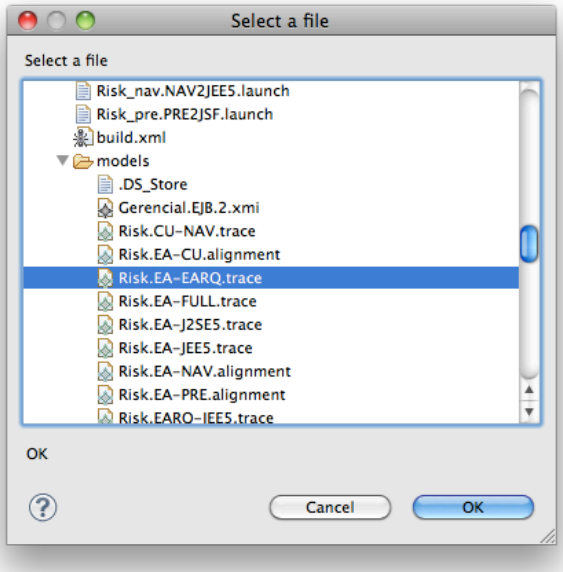


Figure 6.11: Model chooser dialog

In our case study, we choose the `Risk` Business model and Security models. These two models have several corresponding elements that need to be identified by the *application modeler*. Once the models are loaded, it is possible to create the correspondence relationships, identifying elements in both models that represent the same concept in different domains.

C. Creating correspondence relationships

As presented in Section 5.4, at a high-level of abstraction there are 3 kinds of correspondence relationships. For each kind there is a button that adds a relationship in the CM. A relationship is defined by selecting a pair of corresponding elements in the related models.

For instance, in our case study, the *BusinessEntity* `Project` and *ResourceEntity* `Project` represent the same element in two different domains (i.e., Business domain and Security domain). Hence, to define a *CompositionLink* between these pair of elements, it is necessary to choose the *BusinessEntity* `Project` in the main model and the *ResourceEntity* `Project` in the concern model. Once

both elements are selected, the *CompositionLink* relationship can be defined. As presented in Section 4.3.4, at a high-level of abstraction the *application modeler* does not know how exactly these two elements will be composed, he only knows that he wants to compose them in order to protect the information of the entity `Project`. In Figure 6.12 a fragment of the CM for our case study is presented.

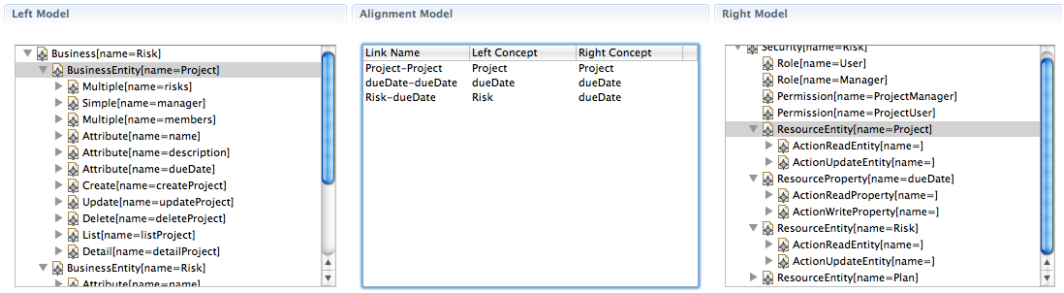


Figure 6.12: Creation of a Compatibility Constraint

In the middle part of Figure 6.12, each correspondence relationship is presented. The left side shows the element in the main model, the right side presents the element in the concern model and the middle shows the type of correspondence relationship between both elements.

Finally, when all the corresponding elements are identified and the relationships are specified in the correspondence model, the *application modeler* must save the model. This model will be used by the Correspondence Derivation Transformation that was generated by the *MTC developer* to generate correspondence relationships between the low-level models.

6.4.2 Traceability Processor

As presented in Section 4.5.3, each transformation rule produces a tracing model. Additionally, all the tracing models generated by the execution of an MTC must be composed into a single tracing model in order to calculate the *transitive closure* of the traces. In order to assist the *application modeler* in composing the generated tracing models, we implement an Eclipse view plug-in called *Traceability processor*. The plugin has two main functionalities:

1. To compose a set of tracing models.

2. To analyze the composed tracing model and tell the user if the model has ambiguous traces.

If there is any ambiguity in the traces, the correspondence derivation mechanism will not be able to generate the low-level correspondence model. This means that the MTCs are not correctly aligned, and the *MTC developer* must

fix the correspondence derivation model. In the following subsections, we will present the tasks that the *Traceability processor* performs:

A. Composing tracing models

The *application modeler* should select the set of tracing models to compose. These models must conform to the tracing metamodel presented in Section 4.5.1. Each model must be added in the right order, starting by the tracing model that relates the highest-level model with the model that follows. Each model is selected using the trace model selector presented in Figure 6.13.

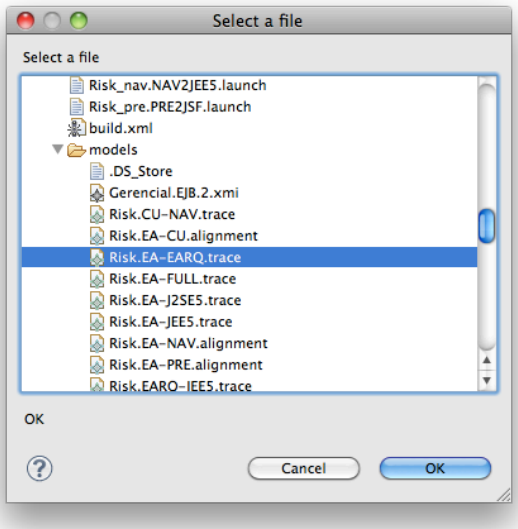


Figure 6.13: Tracing model selector dialog

Figure 6.14 shows the Traceability processor with a set of tracing models loaded. Once all the MTC tracing models are selected, the developer must execute the composition of the models. The composition produces a single model that relates the source elements in the highest-level model with target elements in the lowest-level model. For instance, for the existing MTC, three tracing models where generated: 1) the tracing model between the Business model and the Architecture model (i.e., `Risk.EA-EARCH.trace`), 2) the tracing model between the Architecture model and the JEE model (i.e., `Risk.EARCH-JEE.trace`), and 3) the tracing model between the JEE model and the Java model (i.e., `Risk.JEE-JAVA.trace`). When these three models are processed by the Traceability processor plug-in, a complete tracing model is generated. This complete tracing model is called `Risk.EA-JAVA.trace`.

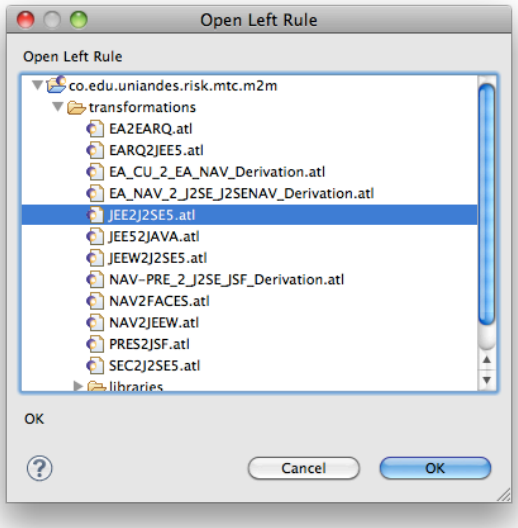


Figure 6.14: Traceability processor view

B. Ambiguity analysis results

Once the composition of the tracing models is executed, the Traceability processor plug-in will inform the developer of the results of the composition. There are three possible types of results:

- *Correct model*: the composed model is correct and can be used without problems
- *Extended CDM required:* the composed model has traces with the same source and several elements that conform to the same metaclass as targets. This means that it is not possible to differentiate between the generated elements using the basic CDM. In this case, it is necessary to use an extended CDM where the compatibility constraints are extended with filters containing information about the transformation rule and output variable that generates the elements.
- *Tracing model cannot be used:* in this case, there are several traces that share the same source, have targets that conform to the same metaclass and are additionally tagged with the same transformation rule name and output variable. In this situation, the *MTC developer* must check the transformation rules to identify the reason of this problem. The possible solutions to this problem are fixing the alignment of the MTCs or extending the information stored in the tracing model in order to correctly identify each corresponding element.

6.5 ATL traceability extension

As previously mentioned, tracing information is one of the main requirements to be able to apply our approach. The *MTC developer* should provide the *application modeler* with a mechanism that automatically generates tracing models each time that an MTC is executed.

Depending on the transformation engine used, there are several options to generate tracing models:

- The extension of the existing transformation rules with the tracing generation logic.
- The use of a high-order transformation that automatically adds the tracing generation logic to the existing transformation rules [Jou05].
- The use of tracing mechanisms provided by the transformation engine [JK06].

In the specific case of ATL-VM, we want to avoid the actual modification of the transformation rules. The HOT option creates an overhead problem because every time that the original transformation rules are changed, it is necessary to re-apply the HOT. This means that we have to inject the modified transformation rule into an ATL model, execute the HOT, and extract the ATL code from the generated model. Moreover, the modification of the HOT for adding customized tracing information, or for using a different tracing metamodel is a complex task. These are some reasons why we extend the ATL-VM to offer a better access to the tracing information when a transformation is executed.

We propose two methods that offer access to tracing information to the *MTC developer*. The first one is a small extension of the ATL-VM providing richer access to the tracing information during the transformation execution. This richer access can be used with an endpoint rule[2] to generate a tracing model. The second method is based on ATL byte-code adaptation that allows for automatically serializing the existing internal tracing model together with the target model. This method has minimal impact on the performance of the ATL-VM, and offers a user-friendly tracing functionality to the transformations developer.

6.5.1 Runtime read access to the tracing information

One possibility to create a tracing model is to have read access the ATL implicit tracing mechanism at the end of the transformation and to recreate the

[2] A *called* rule automatically executed at the end of the transformation.

tracing links in a new model. Therefore, we extend the ATL native types implementation to offer an improved access to the implicit tracing mechanism.

This method offers two main advantages over the HOT method. First, the possibility of adding a *superimposed* module [Wag08] to almost any existing transformation rule to generate a tracing model, keeping the original rules and the tracing-specific rules nicely separated in two different modules. Second, the rules used to generate this model are simpler and easy to change when a customized tracing metamodel is used. However, the drawback of this method is its poor performance. The cause behind this is the necessity to create a copy of the internal tracing model as a final step of the transformation.

6.5.2 Automatic storing of the tracing information

The performance drawback of the previous method is caused by the necessity to iterate and recreate the tracing model. A better strategy, is to directly export the implicit tracing model, minimizing the impact in the execution performance of the transformation rules. This method is the most end-user friendly, because no extra tracing rules are required. The *MTC developer* only needs to activate the generation of the tracing model in the launch configuration. The launch configuration dialogs are presented in Figure 6.15. Once the transformation is executed, the tracing model is stored in the selected path.

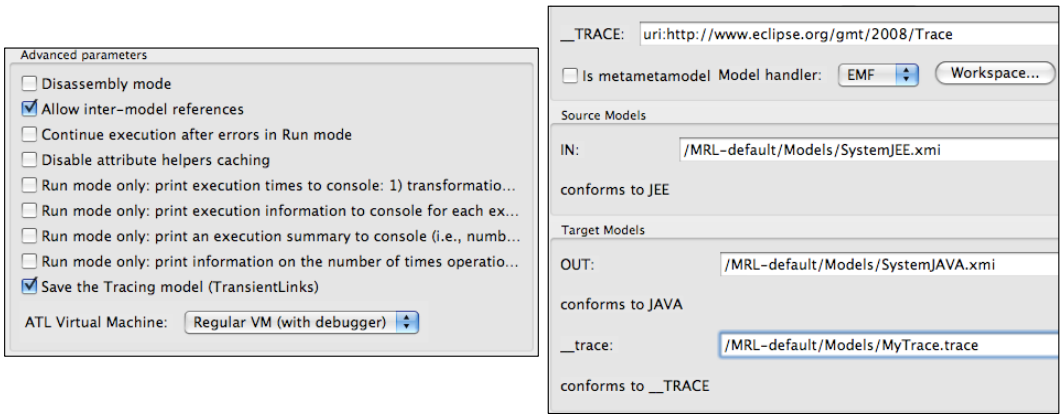


Figure 6.15: ATL Launch configuration dialogs

This method is based on byte-code adaptation similar to the way that *superimposition* is implemented in ATL, and is based on three steps: 1) tracing metamodel and model loading, 2) bytecode adaptation and 3) tracing model serialization.

The main advantages of this method are that it does not require changes to the original transformations, it is end-user friendly and it has low performance overhead. Although the *__TRACE* metamodel and the output

_ _ *trace* model cannot be customized by the end user, they offer the same rich access to the implicit tracing mechanism as the method presented in Section 6.5.1.

6.6 Summary

In this chapter, we have described the architecture of the MTC Framework Toolkit and how it can be used by the *MTC developer* and the *application modeler* to perform various tasks. The MTC Framework Toolkit consists of six Eclipse plug-ins that are used during the alignment of two MTCs as well as in the specification and generation of applications. The *MTC developer's* tasks cover the different steps of the alignment of the two MTCs with a CDM and the generation of the necessary transformations in order to derive the correspondences and to compose/check the generated models. The *application modeler's* tasks cover the different steps of the definition of correspondences between the high-level models and the execution of the multiple transformations until the final application code is generated. The limitations and directions for future work of the MTC Framework Toolkit are discussed in Section 8.5.2.

Chapter 7

Validation: Evolving Transformation Chains

7.1 Introduction

This chapter presents the validation of our approach, which uses a complex case study that adds three Model Transformation Chains (MTCs) to an existing MTC. This case study is quite challenging since it tries to align multiple MTC's at once, instead of the case study used in the text, which was limited to aligning two MTC's. The case study used in this chapter evolves the *Business MTC* presented in Section 3.2 and adds: 1) the *Security MTC* presented in Section 4.3, 2) a *Navigation MTC* that allows to specify Web navigation scenarios using concepts of the navigation domain, and 3) a *Presentation MTC* that allows the specification of a Web user interface using high-level presentation concepts. These four MTCs are aligned using Correspondence Derivation Models (CDMs) between them. These CDMs are used to derive correspondence relationships between the low-level models that are generated by the four MTCs. Finally, the low-level models are integrated using the derived correspondence relationships in order to produce a secure Web application.

This case study is analyzed in order to evaluate if our approach fulfills the key criteria defined in Section 3.3.1. As present above, when a new set of requirements is added to an existing MTC which cannot be specified using the existing assets, it is beneficial to use an approach that preserves the original MTC artifacts. In Section 3.3.1 we present a set of key criteria used to compare between several possible strategies. Furthermore, we will use these key criteria to analyze how the addition of new concern-specific MTCs affect the existing MTC using our approach.

Finally, based on the evaluation of the key criteria, we will analyze if our approach achieves the goals presented in Section 1.2 and overcomes the problems that affect the direct extension of an existing MTC.

In Section 7.2, we present the case study that we will use to validate our

work. Section 7.3 presents the analysis of the key criteria using the case study. Next, in Section 7.4, we show how our approach achieves our research goals. Finally, we discuss the limitations of our work in Section 7.5

7.2 Case Study: 4 Aligned MTCs

In order to perform the validation of our approach we will use the addition of three new MTCs to the existing MTC presented in Section 3.2. As previously introduced, this MTC generates JEE code that supports basic CRUD functions for any business entity defined in a high-level Business Model. Three more concern-specific MTCs are added to this existing MTC in order to extend the functionalities of the original generated applications. These three extra MTCs illustrate several situations that allow us to demonstrate the advantages and limitations of our approach.

This validation demonstrates how we can add new MTCs without disturbing the original assets (i.e., metamodels, models and transformations). These four MTCs are used to generate secured Web applications. The new concern-specific MTCs are: 1) the Security MTC, 2) the Navigation MTC, and 3) the Presentation MTC. Figure 7.1 shows a schema of the 4 interoperating MTCs that are aligned by our Correspondence Derivation Mechanism. In the upper section of the figure presents the 4 MTCs and the derivation relationships. In the lower section of the figure presents the integration of the models that are generated by the MTCs and the code generation step.

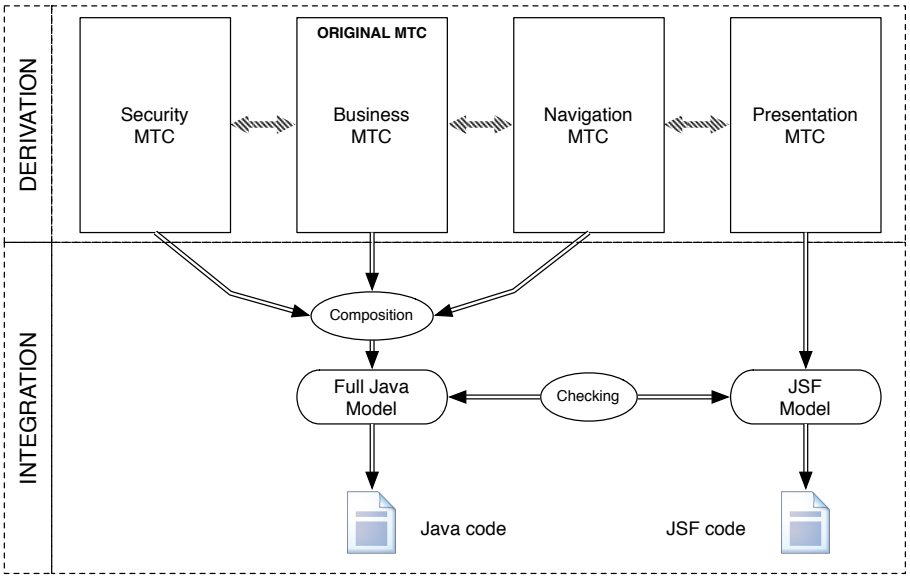


Figure 7.1: Four interoperable MTCs

On the one hand, the Business, Security and Navigation MTCs generate

Java models that conform to the same Java metamodel. These three models are composed and a *full* Java model containing all the business, security and navigation requirements is obtained. On the other hand the Presentation MTC generates a *JavaServer Faces* (JSF) model that conforms to a JSF metamodel. A consistency check between the full Java model and the JSF model is performed and, if successful, the Java and JSF code is generated.

7.2.1 Business MTC

This is the existing MTC which uses a high-level business model to specify JEE applications. The high-level business model conforms to a Business metamodel that defines an application in terms of *BusinessEntities*, their *Attributes*, the *Services* that they provide, and the *Relationships* between them. The high-level model is processed by three model-to-model transformations that bring the business concepts to the technological platform level, where the application is expressed in terms of Java concepts. Additionally, in these three transformations several implementation details are added to the model. Figure 7.2 shows the Business MTC and its four levels of abstraction.

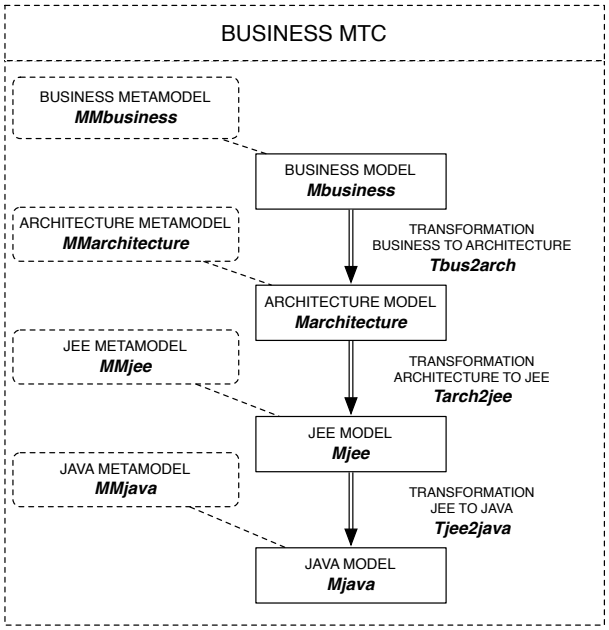


Figure 7.2: Business MTC

7.2.2 Security MTC

The first added MTC allows to specify authorization policies and to generate the security enforcing mechanisms in Java in order to protect the generated

applications. This MTC was presented in Section 4.3 and uses a high-level security model as input. Authorization policies are specified in this high-level security model using high-level authorization concepts. Subsequently, the MTC translates these high-level concepts into Java annotations that are the JEE mechanism to enforce authorization policies. This MTC produces Java *Classes*, *Methods* and *Attributes* with security annotations such as *@DeclaredRoles* and *@RolesAllowed*.

As presented in Chapter 4, this MTC uses a high-level metamodel based on the *SecureUML* metamodel [LBD02]. Using this metamodel, a high-level security model is defined where the authorization policies of an application are independently specified. This high-level security model is processed by a one-step model-to-model transformation that produces a low-level model that conforms to a Java metamodel. This Java metamodel is the same one that is used by the existing Business MTC. Figure 7.3 shows the Security MTC and its two levels of abstraction.

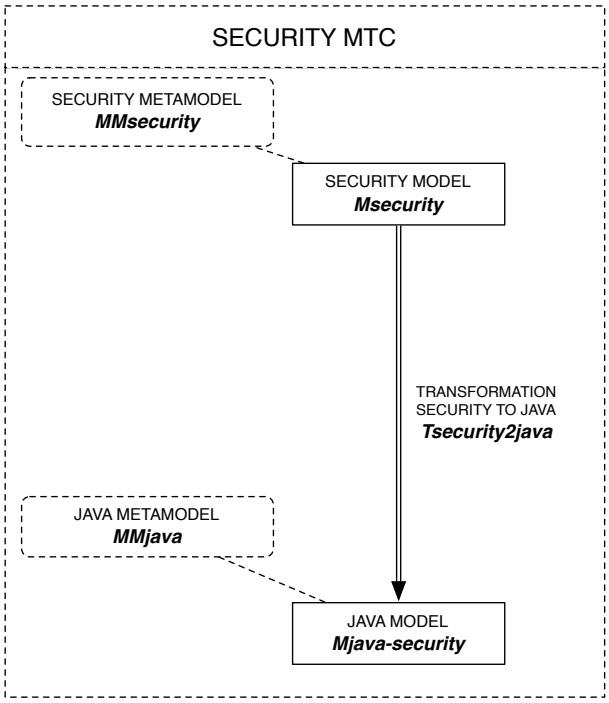


Figure 7.3: Security MTC

7.2.3 Navigation MTC

Even though the original MTC was able to generate the JEE application's business logic, it did not allow to specify user navigation scenarios and to generate the required Java code to control the specified navigation. An additional

functionality that we want to include in the MTC, is the ability to specify customized user navigation scenarios for each application.

A navigation specific MTC is added, offering navigation specific concepts in order to allow to the *application modeler* to specify user navigation scenarios. This MTC has a high-level navigation metamodel with concepts that belong to the navigation domain. These concepts can be used by the *application modeler* to define a high-level navigation model in terms of navigable nodes that can be accessed via navigation links. Next, the MTC transforms the high-level navigation model into a JEE metamodel, that conforms to the existing JEE metamodel presented in Section 3.2.4. At this level, the navigation scenarios are expressed in the form of JEE *Backing Beans*[1]. Finally, the JEE model is transformed into a Java model that conforms to the existing Java metamodel. We reuse the original JEE to Java model-to-model transformation with the purpose of translating the JEE concepts to Java. Figure 7.4 shows the Navigation MTC and its three levels of abstraction.

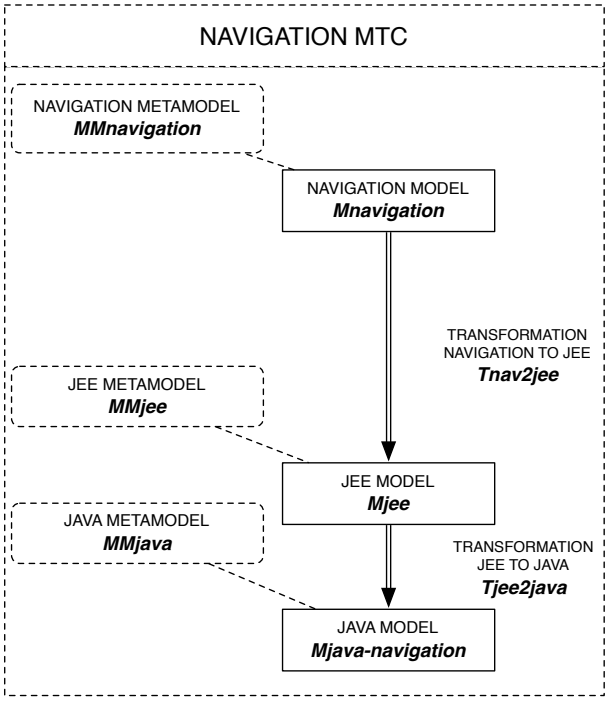


Figure 7.4: Navigation MTC

[1]A JEE Backing Bean represents the controller bean of a use case and is directly referenced by JSF pages.

Navigation metamodel

The *Navigation metamodel* is defined using platform independent Web navigation concepts. This metamodel contains concepts that allow the representation of multiple navigations paths of a Web application. The most important concepts are: *NavigationFlow*, *NavigationClass*, *ProcessClass* and *Link*. A *NavigationFlow* represents a use path between navigation nodes. A *NavigationClass* represents navigable nodes that allow information retrieval or input from the presentation layer. A *ProcessClass* represents a navigation node where information is processed by the application. A *Link* represents a path that interrelates the two navigation nodes. Figure 7.5 shows the Navigation metamodel. Additional details of this metamodel are presented in Appendix A.6.

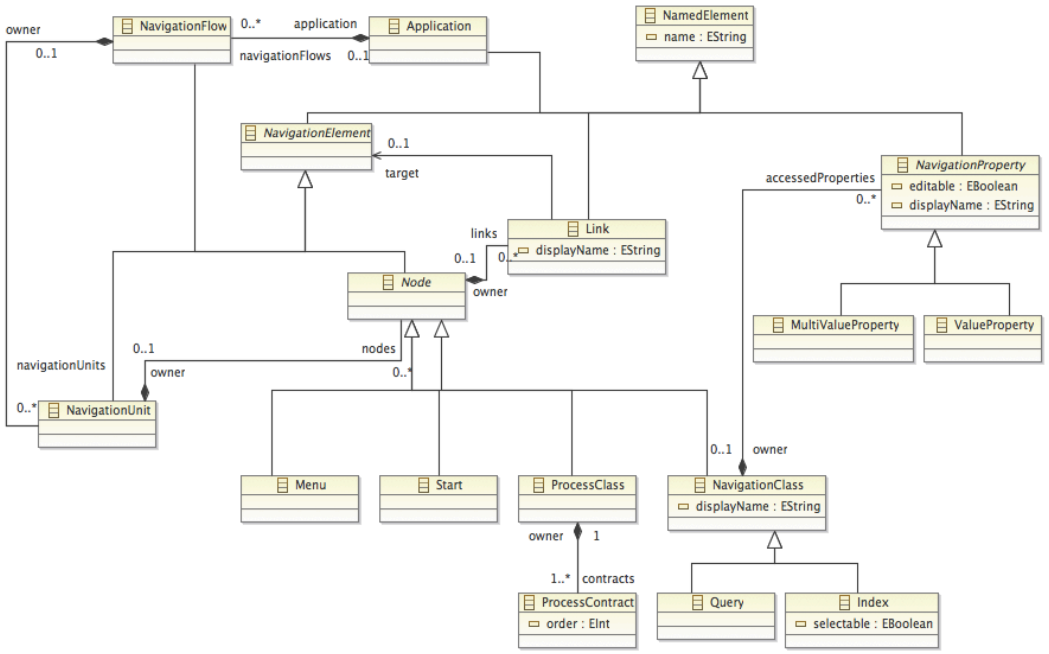


Figure 7.5: Navigation Metamodel

Navigation Model

Figure 7.6 (top) presents a schema of the `Risk` navigation paths between two Web pages. The application starts with a list of `Projects`. From that point, it is possible to modify a `Project` by accessing the update project page. If the action throws an error, the update page is presented again. If the update was successful, the application returns to the list of the `Projects` page.

Figure 7.7 presents a fragment of the `Risk` navigation model, which contains the specification of the *update project* use case scenario. The scenario starts in the *Index* node `ListProject`. This node offers a list of registered `Projects`

showing their `id`, `names`, and `dueDates`. The user can then choose the `UPDATE` *Link* that takes him to the *NavigationClass* `UpdateProject`. This node allows modifying the `Project` *NavigationProperties* (e.g., `name` and `dueDate`). Once the user modifies the information, he can *execute* the *ProcessClass* `UpdateProject`. This *ProcessClass* performs the actual modification of the chosen `Project`. If the process is successful (i.e., `OK`) the user returns to the *Index* node. If the process produces an `ERROR` (e.g., bad-formed date), the application redirects the user to the *NavigationClass* `UpdateProject`, where the details of the error are presented in the *NavigationProperty* `error`.

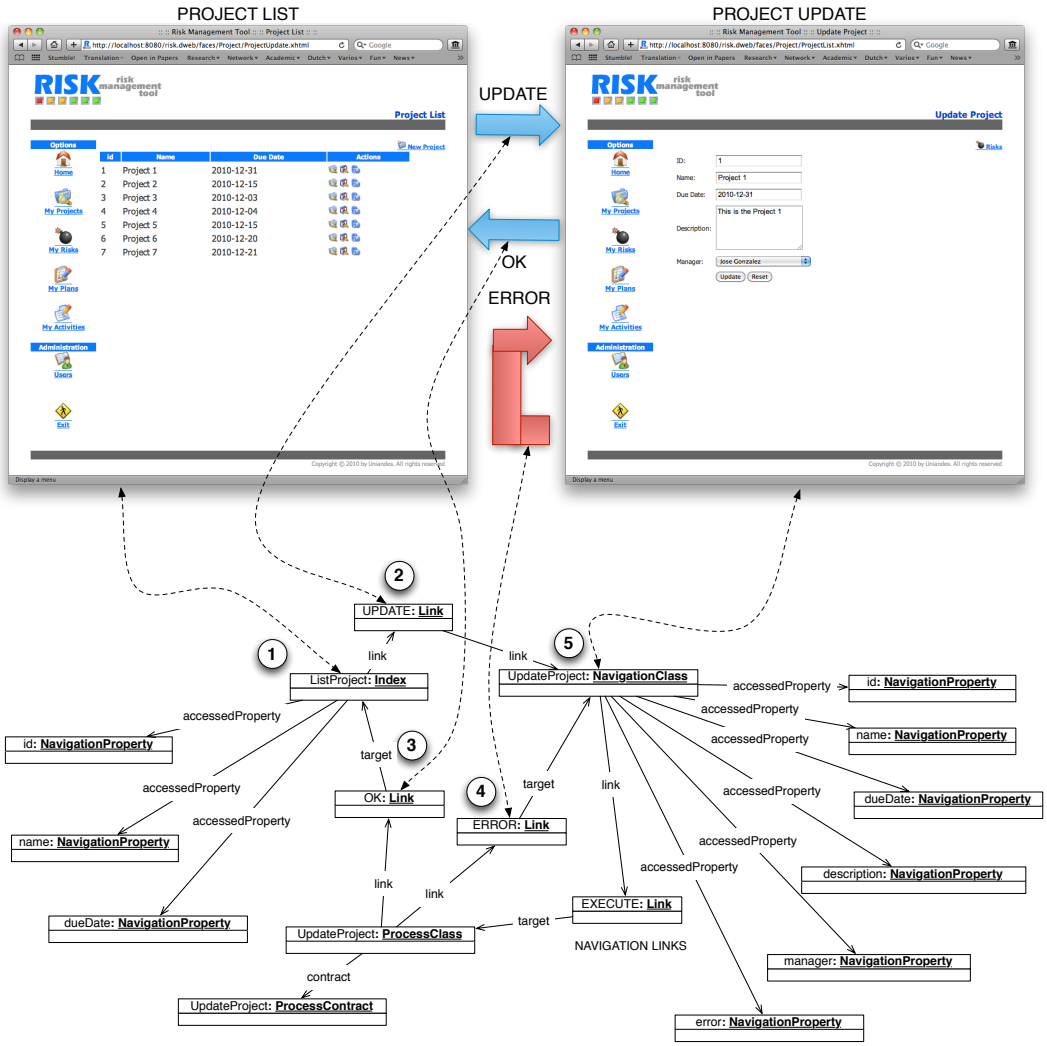


Figure 7.6: Navigation Model

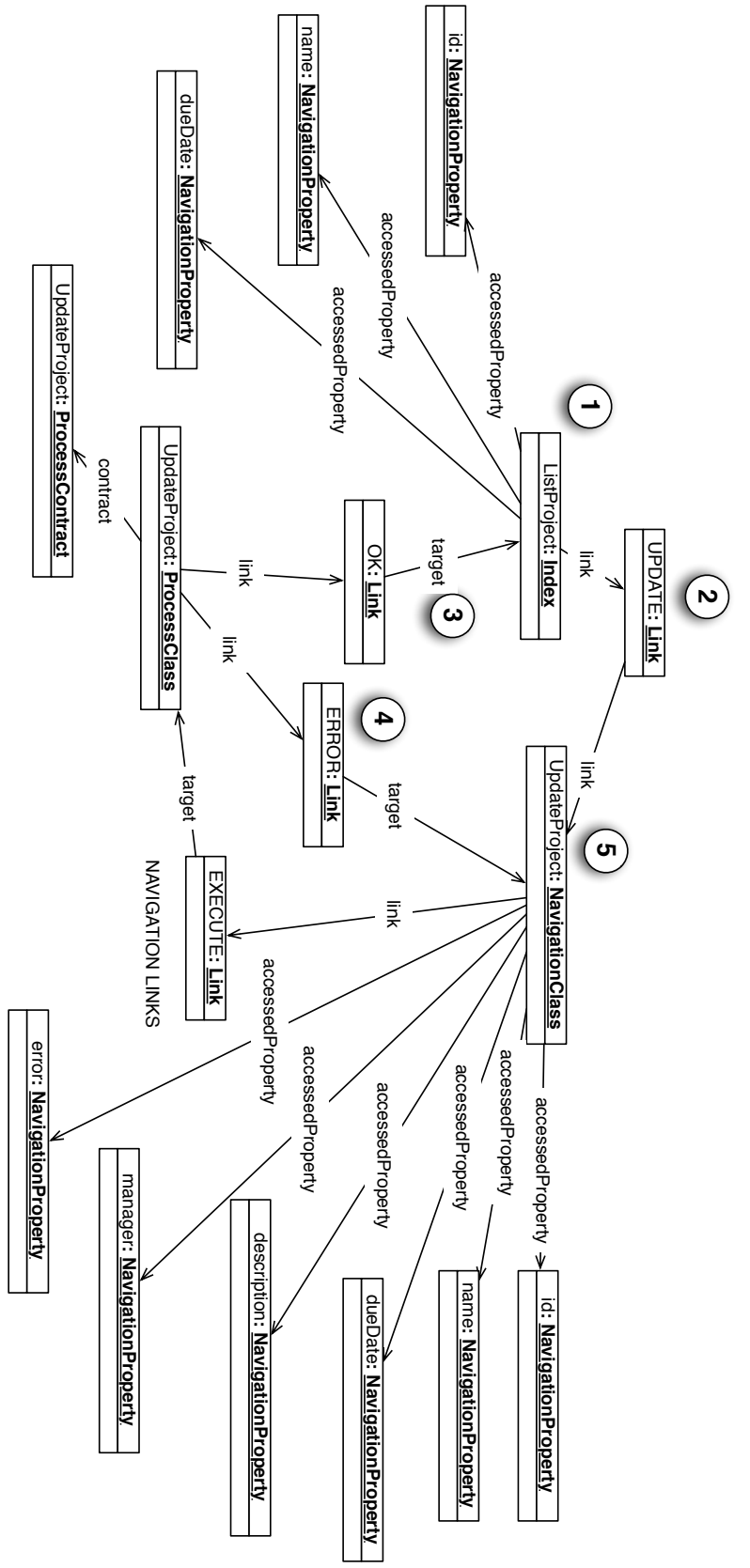


Figure 7.7: Detailed Navigation Model

Navigation to JEE transformation

This model-to-model transformation translates the navigation concepts into JEE concepts that belong to the metamodel presented in Section 3.2.4. For instance, each *NavigationFlow* is transformed into a *Controller* which is a special *BackingBeans* that controls the navigation flow of the application. Additionally, each *NavigationClass* is transformed into a *BackingBean* that allows the presentation layer to access the information of the entities, into a *BusinessDelegate* and into a *BusinessObject*. The *BusinessDelegate* and the *BusinessObject* are elements that are already generated in the Business MTC and will be composed with the elements that are generated by the Navigation MTC.

JEE to Java transformation

The final model-to-model transformation translates the JEE concepts into low-level Java concepts, as presented in Section 3.2.7. In this case, this transformation is responsible for translating the *Controllers*, *Backing Beans*, the *BusinessObjects* and *BusinessDelegates* into Java classes. It is important to notice that only the classes generated by the *BusinessObjects* and *BusinessDelegates* will be composed.

7.2.4 Presentation MTC

As in the case of the application's navigation, the original MTC did not allow specifying the Web user interface, and it did not support the generation of the Web layer and JavaServes Faces (JSF) [CS09] pages.

The purpose of adding a presentation specific MTC is to provide a mechanism to the *application modeler* in order to specify and generate customized Web interfaces. The Presentation MTC offers a metamodel with high-level Web user interface concepts to the *application modeler*, which can be used to specify the structure of a particular presentation for each application.

The Presentation MTC offers a Presentation metamodel that allows the *application modeler* to define the structure and content of the Web interface. The *application modeler* can use the platform independent presentation concepts in order to define the different *Pages* that are provided to the final user. The content included in each *Page* is specified by using *UIControls*. Then, the high-level presentation model is transformed by the Presentation MTC into a JSF model. Figure 7.8 shows the Presentation MTC and its two levels of abstraction. The generated JSF pages need to access the information that is provided by the business logic through the *backing beans*. The *backing bean* classes are generated by the Navigation MTC. Therefore, the Presentation MTC must be aligned with the Navigation MTC.

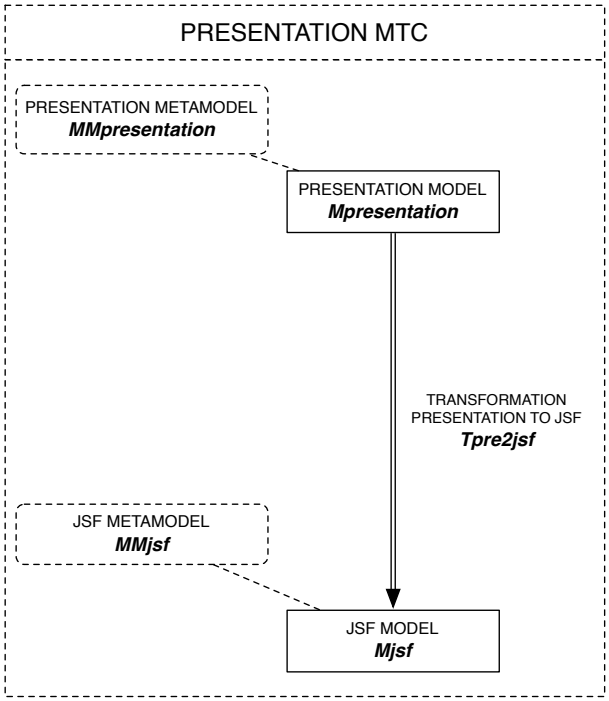


Figure 7.8: Presentation MTC

Presentation metamodel

The *Presentation metamodel* is defined using platform independent Web presentation concepts. This metamodel contains concepts that allow to represent the different Web pages of a Web application and their respective content. The main concept is *Page*, which represents a Web page. Each *Page* can have multiple *Views* that represent the multiple simultaneous views of a page. In addition, each view can be associated with a *PresentationClass*. A *PresentationClass* is an element that represents content that is provided by the application logic. Each *PresentationClass* should be connected to a *NavigationClass* in the high-level navigation model. A *PresentationClass* contains a set of *UIElements* that represent the Web user interaction components that displays and captures information. The *UIElements* are specialized in several metaclasses, such as: *TextInputs*, *Tables*, *Buttons*, etc. Figure 7.9 shows the Presentation metamodel. Additional details of this metamodel are presented in Appendix A.7.

Presentation Model

Figure 7.10 (top) shows a `Risk` Web page that presents the list of projects. Figure 7.10 (bottom) presents a `Risk` presentation model fragment that represents this Web page. A page that has a table with the list of projects is

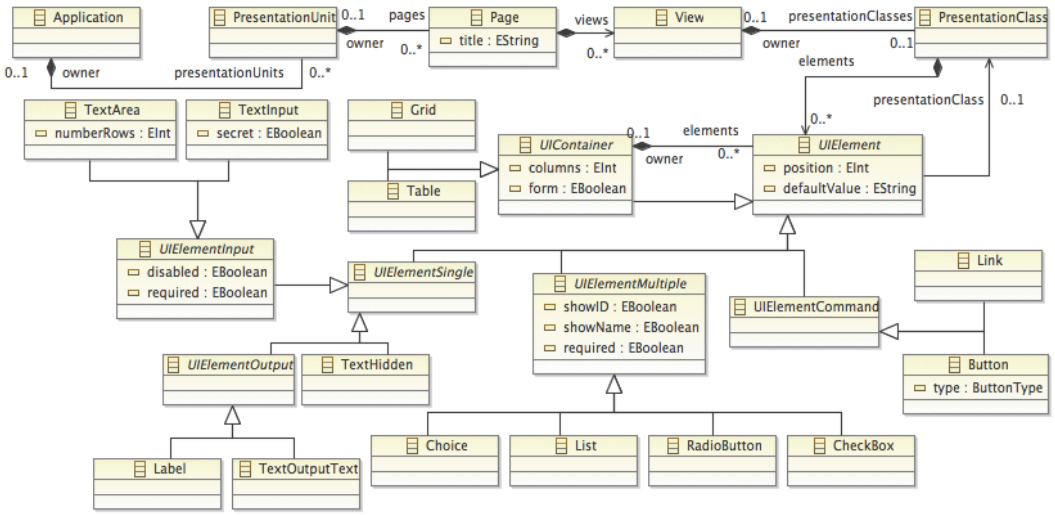


Figure 7.9: Presentation Metamodel

specified in this fragment. The *Page* `ProjectList` has a `View`, and the *View* has a *PresentationClass*. The *PresentationClass* `ProjectList` must be related by a correspondence relationship with a *NavigationClass* in the high-level navigation model. Additionally, the *PresentationClass* `ProjectList` contains a *Table* with four *Columns*: 1) a *Column* with the `id` of the `Project`, 2) a *Column* with the `name` of the `Project`, 3) a *Column* with the `dueDate` of the `Project`, and 4) a *Column* with three *Buttons*. These *Buttons* will call the services offered by the *BackingBeans* in the navigation model. The services called are: `DetailProject`, `UpdateProject` and `DeleteProject`.

JSF metamodel

The *JSF Metamodel* is defined using the concepts of the technological platform *JavaServer Faces* (JSF). This metamodel contains concepts that allow the representation of different JSF pages and their user interface components. For instance, to present information, to capture data and to perform advanced interaction with the end-user. The main concept is *Page* that represents a JSF page. A *Page* is included in a *Folder* and contains *UIElements* that are the user interface components. There are several types of components such as: *Forms*, *TextFields*, *Tables*, *Buttons*, etc. Figure 7.11 shows the JSF metamodel. Additional details of this metamodel are presented in Appendix A.8.

Presentation to JSF transformation

This model-to-model transformation translates the platform independent Web presentation concepts into low-level JSF concepts. Generally speaking, this is a simple transformation where most of the elements are directly transformed

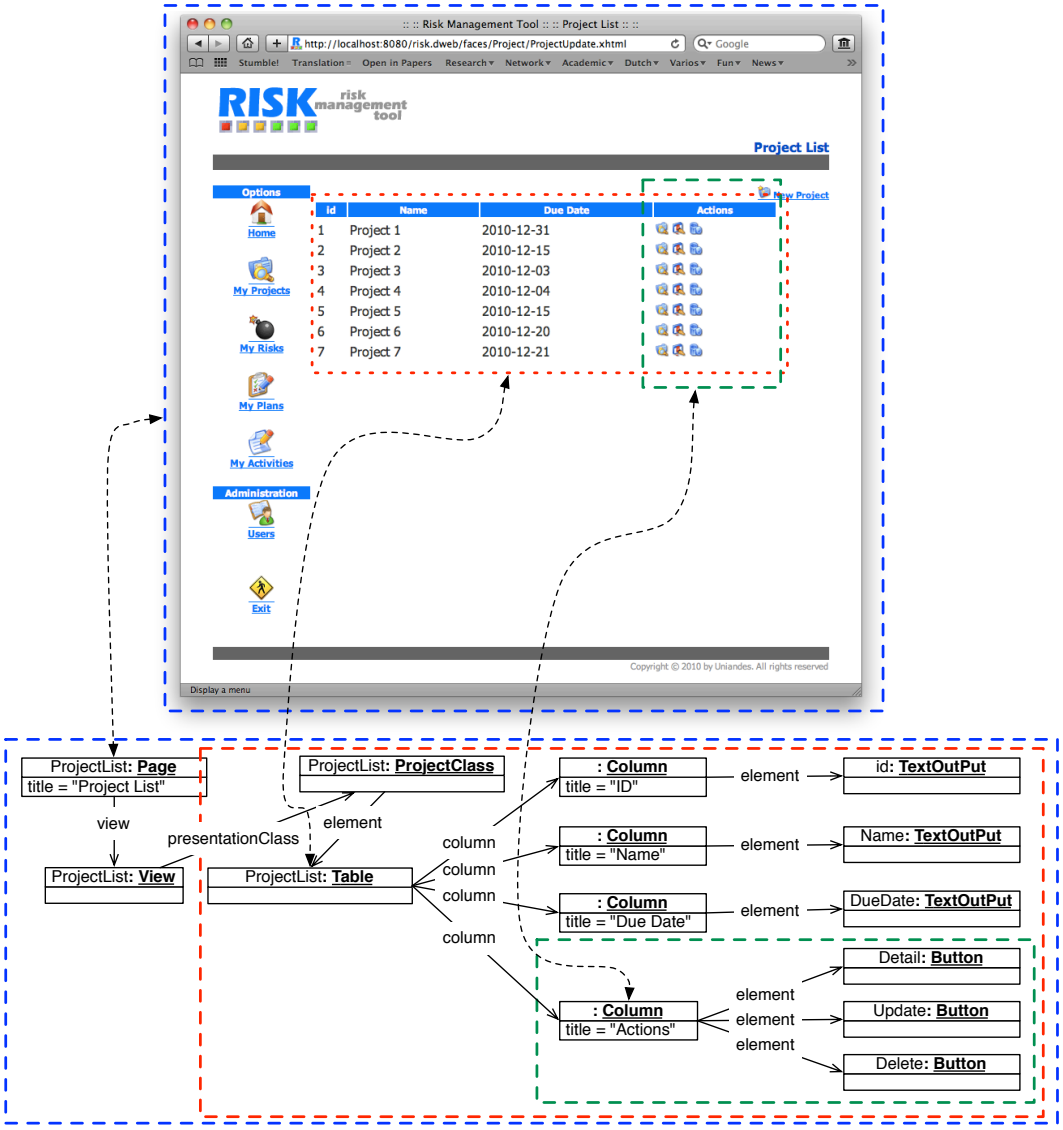


Figure 7.10: Presentation Model

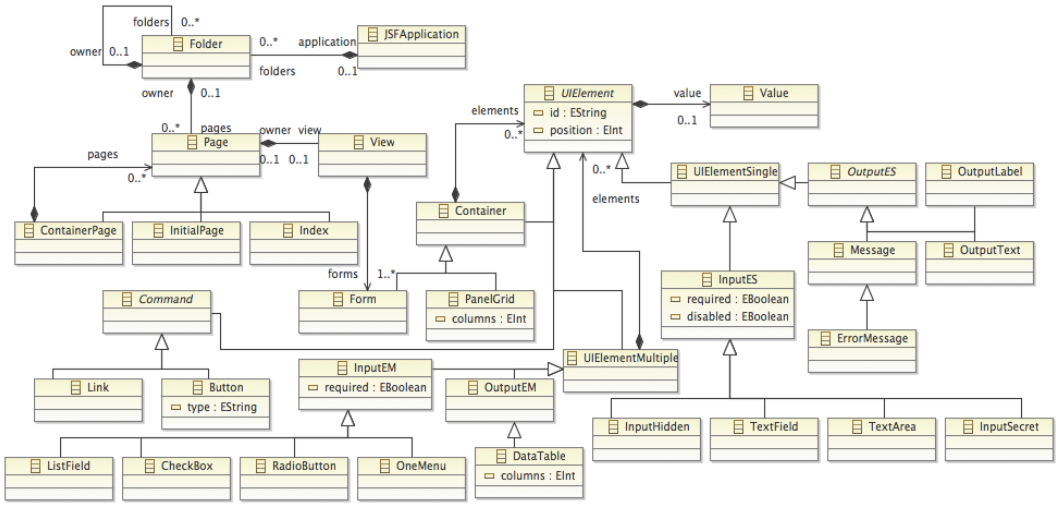


Figure 7.11: JSF Metamodel

to JSF concrete concepts. Additionally, the transformation adds some JSF specific details to the target model.

7.2.5 High-level correspondence models

Once the four MTCs are implemented and aligned by the *MTC developer*, and the four high-level models are defined by the *application modeler*, the *application modeler* must identify the corresponding elements between these high-level models. Three correspondence models are needed in this case study:

1. A CM between the business model and the security model (CM_{b-s}).
2. A CM between the business model and the navigation model (CM_{b-n}).
3. A CM between the navigation model and the presentation model (CM_{n-p}).

Figure 7.12 shows the four high-level models and the three correspondence models.

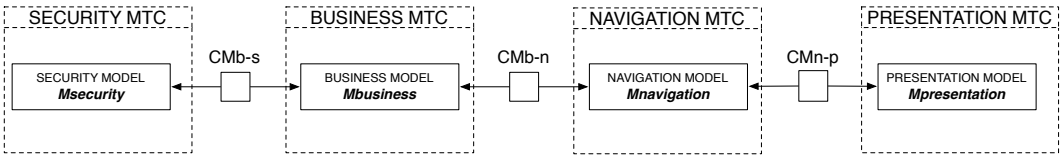


Figure 7.12: High-level Correspondence Models

Business/Security Correspondence Model

The CM that is defined between the Business high-level model and the Security high-level model CM_{b-s} was described in Section 5.3.1. This CM relates the protected *Resources* in the security model with the *BusinessEntities*, *Attributes* and *Services* in the business model.

Business/Navigation Correspondence Model

To extend the Business MTC functionality with the functionality offered by the Navigation MTC, we need to identify the corresponding elements between the Business high-level model and the Navigation high-level model.

As previously explained, multiple usability scenarios are modeled in the navigation model. The scenarios are specified in terms of navigation nodes and the links between them. We use correspondence relationships to identify which entities offer or receive information as navigation nodes.

The high-level correspondence model (CM_{b-n}) is defined between the `Risk` business high-level model and its navigation high-level model. This CM contains relationships between three pairs of elements: 1) *BusinessEntities* with *NavigationClass*, 2) *Attributes* with *NavigationProperties* and 3) *Services* with *ProcessContracts*. These pairs represent the same concept in different domains. For instance, the *BusinessEntity* `Project` is the base for a series of navigation scenarios where a `Project` is created, updated, and listed. Consequently, all the *NavigationClasses* that access or modify the *BusinessEntity* `Project` must be related by a correspondence relationship with this *BusinessEntity*.

Navigation/Presentation Correspondence Model

The high-level correspondence model (CM_{n-p}) that is defined between the `Risk` navigation model and its presentation model relates the navigation nodes with the Web pages that present and capture information from the final user. This CM has relationships between: 1) a *PresentationClass* in the presentation model and a *NavigationClass* in the navigation model, and 2) a *UIElement* in the presentation model and a *NavigationProperty* or a *ProcessContract* in the navigation model. The *UIElements* will show to the final user the information provided by the *NavigationProperties* or execute *ProcessContracts*.

7.2.6 Correspondence relationships derivation

As explained in Section 4.4, there are two requirements to obtain a low-level correspondence model: 1) tracing back the targets that are generated from corresponding sources, and 2) constraining the generated correspondence relationships. The first requirement (i.e., tracing models) is automatically generated by the extension that we did to the ATL-VM. In order to fulfill the

second requirement, the *MTC developer* needs to define a CDM for each pair of MTCs. In this case we need three CDMs: 1) Business/Security CDM, 2) Business/Navigation CDM, and 3) Navigation/Presentation CDM. Figure 7.13 presents a detailed schema of the correspondence relationships derivation for the four MTCs.

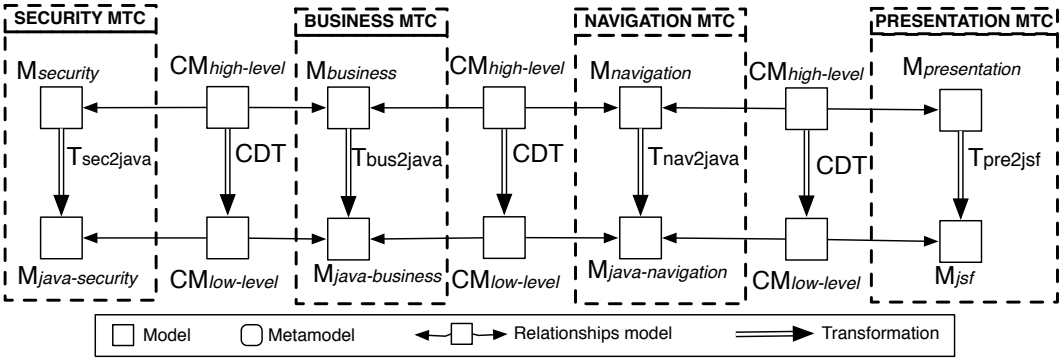


Figure 7.13: Correspondence Relationships Derivation

Deriving the Business/Security low-level correspondence model

Chapter 4 explained the derivation of the correspondence relationships between the lowest level models that are generated by the Business MTC and the Security MTC.

Deriving the Business/Navigation low-level correspondence model

The *MTC developer* needs to define a CDM between the lowest level metamodels in both MTCs in order to derive the correspondence relationships between the lowest level models that are generated by the Business MTC and the Navigation MTC

In this case both lowest level models conform to the Java metamodel, which allow us to perform a homogeneous composition between them.

As was mentioned in Section 7.2.3, the Navigation MTC produces a model which contains several classes. These classes are the actual Java implementations of *BackingBeans*, *BusinessObjects* and *BusinessDelegates*. In this scenario, the *BackingBeans* are completely generated by the Navigation MTC, but the *BusinessObjects* and the *BusinessDelegates* are "place holders" that will be replaced by the "real" ones that are generated by the Business MTC. In other words, the correspondence derivation mechanism must create correct matches between the Java classes of the *BusinessObjects* and the *BusinessDelegates* in both models.

Deriving the Navigation/Presentation low-level correspondence model

The Navigation MTC produces a model that conforms to the Java metamodel, and the Presentation MTC produces a model that conforms to a JSF metamodel. At the code-level, Java and JSF are dynamically composed by the Application Server (e.g., GlassFish, JBoss). Therefore, we do not need to compose them at the model-level. However, we need to verify if the generated models are consistent, or in other words, if the elements that are called in the JSF Pages are present in the Java model.

The correspondence derivation mechanism should identify which elements in the navigation model are called or accessed by the elements in the JSF model. For instance, the CDM must contain a relationship between the *DeclaredClass* metaclass in the Java metamodel and the *Page* metaclass in the JSF metamodel. This will generate a low-level correspondence relationship between each generated *Page* and the Java class that implements the *BackingBean*.

7.2.7 Integrating the MTCs

The correspondence relationships between the 4 lowest level models must be resolved to produce a complete application. On the one hand, a homogeneous composition can be performed between the models generated by the Business MTC, the Navigation MTC and the Security MTC, which conform to the Java metamodel. On the other hand, a consistency checking can be executed between the models generated by the Navigation MTC, and the Presentation MTC, which conform to different metamodels.

In the case of the homogeneous composition that will be performed between the three Java models, the composition order does not affect the final result. This is due to the fact that the corresponding elements between the business low-level model and the security low-level model differ from the corresponding elements between the business low-level model and the navigation low-level model. This means that we have two options to compose the three models: 1) we can compose the business low-level model with the security low-level model, and then the composed model with the navigation low-level model, or 2) we can compose the business low-level model with the navigation low-level model, and then the composed model with the security low-level model. In addition, the security low-level model cannot be directly composed with the navigation low-level model due to the fact that there are no direct correspondence relationships between these two models.

In the case of the consistency checking between the navigation low-level model and the presentation low-level model, we can perform this check before or after the business/navigation composition. The only requirement is that if the composition is performed before, a new set of correspondences must be

derived between the composed model and the presentation low-level model.

Finally, after performing the composition of the three java models and checking the presentation low-level model, we reuse the existing model-to-text transformation that generates the Java code from the composed Java model. For the presentation low-level model (i.e., JSF model) a JSF specific model-to-text transformation is used.

Figure 7.14 presents the composition execution order: 1) Business/Security composition, 2) Secured Business/Navigation composition, 3) full Java model/Presentation checking, and finally 4) the Java code and JSF code generation step.

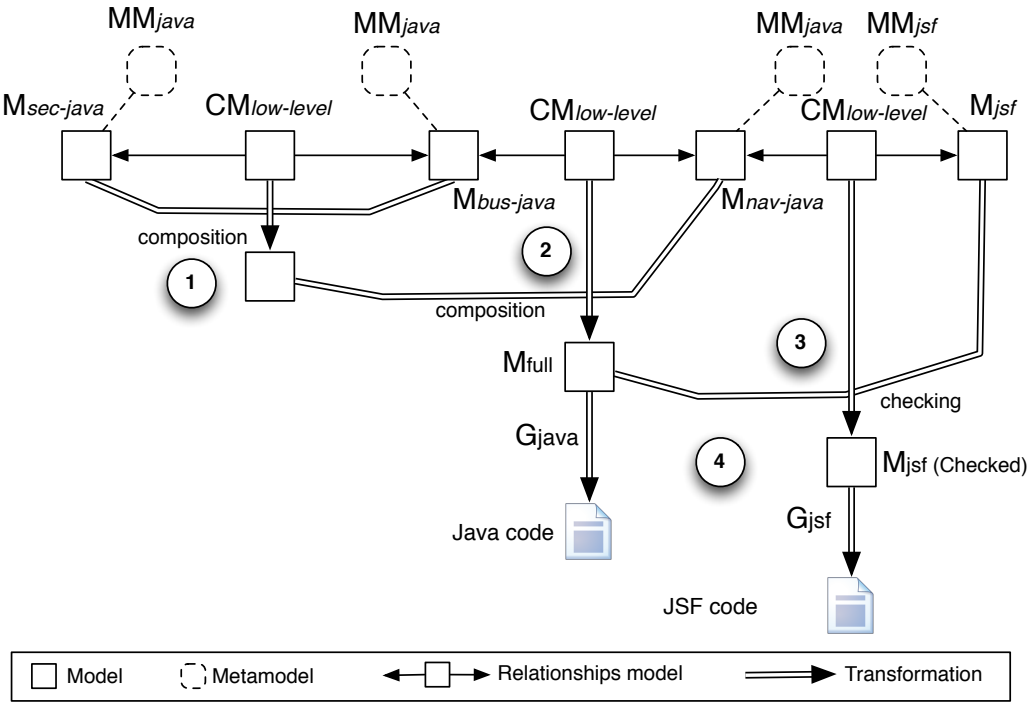


Figure 7.14: Integration of the MTCs

1. Business/Security Composition

As previously described in Chapter 5 the Java model ($Msec-java$) generated by the Security MTC is integrated with the Java model ($Mbus-java$) generated by the existing Business MTC. The integration of these two low-level models takes advantage of the homogeneous nature of the common low-level Java metamodel. This allows the reusing of the composition mechanism that we implemented for the Java metamodel. The correspondence model between these two low-level models is used to perform this composition. The details of

this composition were presented in Chapter 5. The product of this composition is a secured Java model.

2. Secured Business/Navigation Composition

Before preforming the second composition between the composed model, which was generated in the previous step, with the Java model that was generated by the Navigation MTC, we need to derive a new CM between these two models. This correspondence derivation is a simple task since the composition copies every element in the business java model into the composed model. Additionally, it adds the elements of the security java model to the composed model. Therefore, for each corresponding element in the business java model, the copied element in the composed model will have a corresponding relationship as well. Our derivation mechanism only needs to follow the traces.

Once the derivation mechanism generates the new CM between the composed model and the Java model, which was generated by the Navigation MTC, we can perform the second homogeneous composition. In this composition the empty Java classes for the *BusinessDelegates* and *BusinessObjects* in the navigation low-level model are composed with the complete Java classes *BusinessDelegates* and *BusinessObjects*, which were generated by the Business MTC and copied to the composed model generated in the previous step.

The new composed model contains all the business entities that were specified in the high-level business model, and for each business entity we offer basic CRUD access to their information. Additionally, this model has security annotations that enforce the authorization policies that were specified in the high-level security model. Finally, this model has all the *BackingBeans* that control the navigation of the application and provide the information required by the user interface.

3. Full Java model/Presentation Checking

Once the full Java model is obtained by composing the models generated by the Business MTC, the Security MTC and the Navigation MTC, we perform a consistency check between the full Java model and the JSF model, which was generated by the Presentation MTC. The objective of this check is to verify that the elements in the JSF model that access the Java application have the correct references to the elements in the Java model. This means that if an element in the Java model has a correspondence relationship with an element in the JSF model, then these two elements must have the same name or have an attribute with the same value. We use our checking mechanism to verify corresponding elements and to report if contains consistency problems.

4. Code generation

Finally, the application code can be produced once the consistency between the full Java model and the JSF model is verified. To produce the Java code we use the model-to-text transformation that belongs to the original Business MTC. This transformation can be directly reused because we did not add any concept to the Java metamodel, and the full Java model conforms the original Java metamodel. Additionally, we need to implement a model-to-text transformation that translates the JSF model into JSF code.

Figure 7.15 presents a detailed schema of all the models involved in our JEE Web applications generator, which is formed by four concern-specific MTCs (i.e., Business MTC, Security MTC, Navigation MTC, and Presentation MTC). The elements in the colored squares are created or generated using our *MTC Framework Toolkit*.

7.3 Key Criteria Analysis

To evaluate our approach we use the key criteria that were presented in Section 3.3.1. These key criteria were chosen to compare different strategies that can be used to evolve an existing MTC. Each key criterion includes a question that allows us to discern if the strategy has a positive or a negative impact in the evolution of an MTC. In addition, each key criterion is related with one or more of our research goals, which were presented in Section 1.2. In this section, we analyze our approach using these key criteria. A summary of this analysis is presented in Table 7.1. The analysis for each criterion applied to our approach is presented below.

7.3.1 Criterion 1 (C1): Impacted artifacts

Measuring the impact of the changes in the MTC artifacts is the most important key criterion that our approach must fulfill in order to evolve an existing MTC. We want to avoid to cause a detriment in the maintainability and understandability of MTC artifacts due to changes to them. Additionally, we want to avoid breaking the multiple dependencies that exist between metamodels and models, metamodels and transformations, and each transformation step and those which follow it. In summary, we want to avoid a ripple effect of these problems throughout the entire MTC, and in addition preserve the original artifacts unchanged.

The original Business MTC artifacts (i.e., metamodels, models and transformations) are preserved using our approach, as three additional concern-specific MTCs are added. These three MTCs were aligned with the existing one using correspondence models and this alignment is kept throughout the

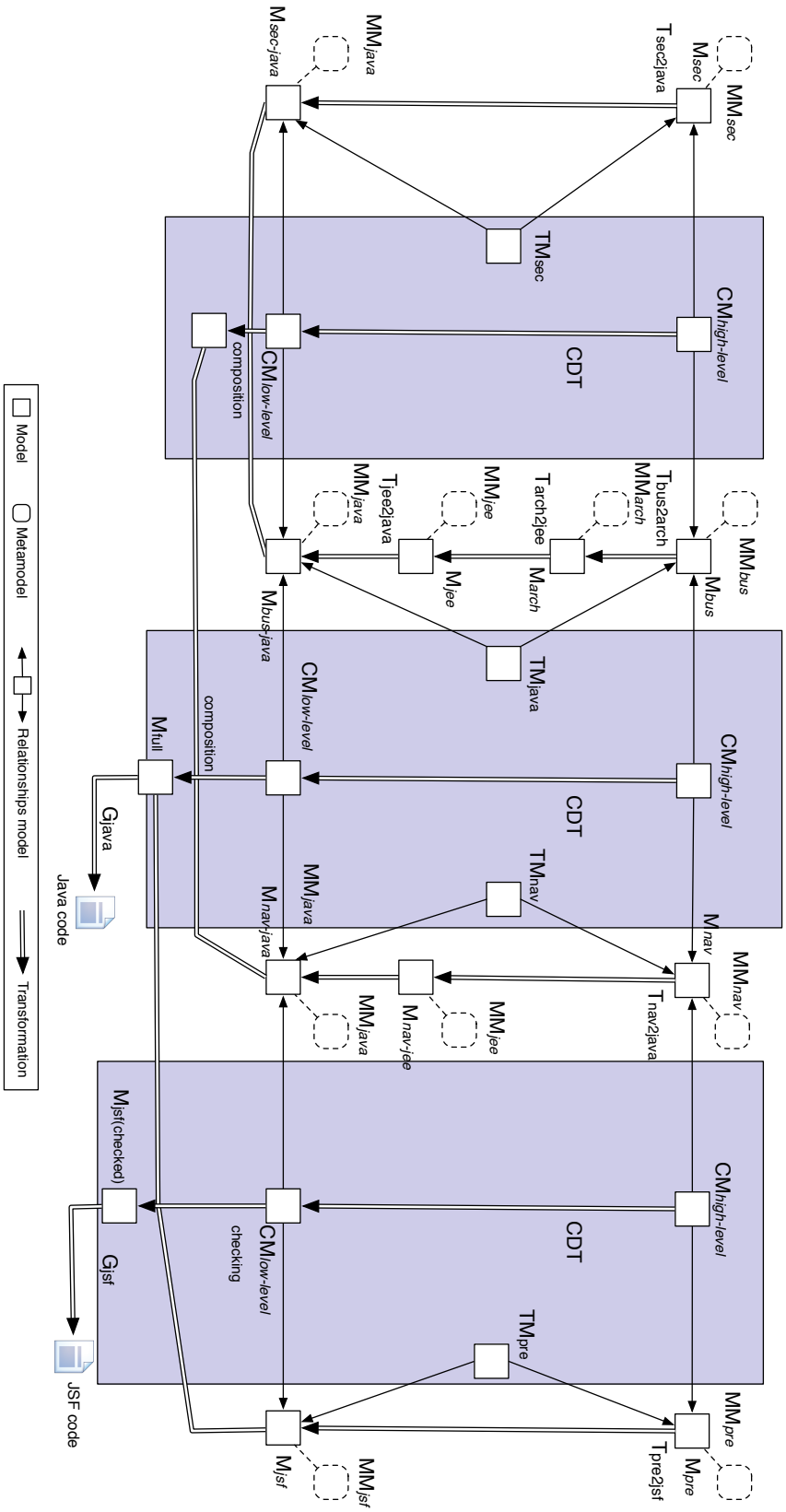


Figure 7.15: Four interoperable MTCs

Criteria	Question	Result
C1: Impacted Artifacts	Do the existing metamodels, models and transformations remain unchanged?	Yes (+)
C2: High-level concern concepts	Are concern-specific concepts available in the metamodel(s) to specify the added concern?	Yes (+)
C3: Metamodel Pollution	Are the metamodels free of alien concepts that do not belong to their domain?	Yes (+)
C4: Monolithic Models	Is a set of concern-specific models used to specify the whole application?	Yes (+)
C5: Number of impacted elements	Are the model elements impacted by the new concern specified at a high-level of abstraction?	Yes (+)
C6: Identification complexity	Is the new concern specified at the same level of abstraction as the elements it affects?	Yes (+)
C7: Integration mechanism	Is it possible to use a common integration mechanism?	Yes (+)

Table 7.1: Analysis of the key criteria

whole set of transformation steps with our correspondence derivation mechanism. Consequently, the three generated Java models are composed by using the derived correspondence relationships. In addition, the JSF model and the composed Java model are checked in order to verify consistency between them. Finally, the full Java model is transformed into code by reusing the original model-to-text transformation, and the JSF is transformed into code by a new model-to-text transformation. In summary, it can be said that the Business MTC is oblivious of the added MTCs.

As we can see, every existing artifact is unmodified and the impact of the changes is minimized. The answer to the question "*Do the existing metamodels, models and transformation remain unchanged?*" is **YES**, the existing metamodels, models and transformation remain unchanged.

7.3.2 Criterion 2 (C2): Use of high-level concern-specific concepts

The use of *Domain-Specific Modeling Languages* (DSMLs) increases the level of abstraction and gives suitable concepts to domain experts to specify an

application close to the problem domain. This means that MTC high-level metamodel(s) should offer concern-specific concepts that allow the *application modeler* to specify the new concern-specific requirements.

We implement each MTC in our case study by using four concern-specific metamodels (i.e., Business metamodel, Security metamodel, Navigation Metamodel and Presentation metamodel). These metamodels, allow the *application modelers* to specify the application using concepts that belong to each concern. For instance, the business experts can express the application structure by using *BusinessEntities*, and their *Attributes*, *Associations* and *Services*. The security experts can specify the authorization policies of the application with the use of concepts such as *Resources*, *Permissions*, *Actions* and *Roles*. The navigation experts can define use case paths by using *NavigationFlows*, *NavigationClasses* and *Links* between them. Finally, the presentation expert can specify the structure of the presentation with concepts such as *Pages* and *UIElements*. *UIElements* are concepts that represent user interface controls that allow the interaction with the final user.

As we can see, every MTC has a DSML that enables the concern experts to specify the application with the use of appropriate concepts. The answer to the question "*Are concern-specific concepts available in the metamodel(s) to specify the added concern?*" is **YES**, the existing metamodels offer concern-specific concepts that allow the domain experts to specify the application using the most appropriate concepts.

7.3.3 Criterion 3 (C3): Metamodel pollution

In order to avoid the *metamodel pollution* problem presented in Section 1.1.1, it is important to avoid adding concepts to a metamodel that clearly do not align with its existing concepts. This is detrimental to the understandability and maintainability of the metamodel.

Four concern-specific metamodels are used in our case study to specify an application. For the purpose of preserving each one of these metamodels independent and oblivious of the others, we avoid to pollute them with concepts that do not clearly align with their concern-specific concepts. For instance, we did not add presentation concepts such as *Page* or *UIElement* in the Business metamodel. On the one hand, the concepts that belong to the Business metamodel are suitable to be used by business experts in order to define the business structure. On the other hand, the concepts that belong to the Presentation metamodel are suitable to be used by the Web user interfaces designer with the purpose of specifying the application Web pages. By maintaining these two metamodels encapsulated, we avoid to pollute the metamodels with alien concepts that are detriment for their maintainability and understandability.

As we can see, each metamodel is independent of the others and contains all the concepts that belong to its concern to specify each high-level model.

The answer to the question "*Are the metamodels free of alien concepts that do not belong to their domain?*" is **YES**, the used metamodels contains only the concepts that belong to the specific concern.

7.3.4 Criterion 4 (C4): Monolithic model

As explained in Section 1.1.1, the use of a *monolithic model* to specify multiple concerns of an application increases the complexity of the model and making it more difficult to comprehend and to maintain. The use of several concern-specific models follows the SoC principle, which reduces the complexity of specifying each concern model.

Each MTC in our case study uses an independent model as input, which conforms to a concern-specific metamodel. In each one of these models, the concern experts specify the application requirements with the use of a suitable language for each concern. These independent models help to comprehend, to specify and to evolve each concern model. In our example, the presentation modeler can specify the multiple Web pages of the application without thinking about the different entities of the application or how they relate between each other. In a similar manner, the security modeler does no need to know which are the navigation paths of the application. The fact that each model is oblivious of the others helps to specify and understand the requirements of each concern.

As we can see, each model is independent of the others and contains only the specification of a concern. The answer to the question "*Is a set of concern-specific models used to specify the whole application*" is **YES**, multiple independent concern-specific models are used to specify an application.

7.3.5 Criterion 5 (C5): Identification of impacted model elements complexity

Identifying the model elements that are impacted by a new concern-specific requirement will be a simpler task at the high-level than at the lowest level. The reason behind this is that at a high-level of abstraction, a large number of implementation details are hidden. This will reduce the amount of work that the *application modeler* needs to perform in order to identify the elements that are impacted by the addition of new concern-specific requirements.

We align the different models at a high-level of abstraction by using a correspondence model. This correspondence model allows the *application modeler* to identify the elements that are impacted by a new concern-specific requirement. For instance, suppose that a new authorization policy for *BusinessEntity* `Project` is required. This new authorization-specific requirement can be specified independently of the existing *BusinessEntity* `Project` and subsequently, a correspondence between them at the high-level of abstraction can be created.

In other words, the correspondence relationships are used as a mechanism to identify the elements that are affected by a new concern-specific requirement. In contrast, if the *application modeler* wants to identify, at a low-level of abstraction, all the classes, methods and attributes that need to be modified in order to add the authorization mechanisms, he will need to posses deep knowledge of the Java platform and the application.

In our case study, we manually identify each element that is affected by a new concern-specific requirement at a high-level of abstraction. The answer to the question "*Are the model elements impacted by the new concern specified at a high-level of abstraction?*" is **YES**, we identify with a correspondence relationship which elements are affected by a new requirement at a high-level of abstraction.

7.3.6 Criterion 6 (C6): Complexity of identifying the impacted model elements

We want to specify the new concern-specific requirements and identify the impacted model elements at the same level of abstraction. If new concern-specific requirements are specified at a different level of abstraction than the existing elements that they impact, then this will require the knowledge of the two involved levels of abstraction (e.g., concern level and platform level). This will increase the *complexity of identifying the impacted model elements*.

We define a correspondence model in our case study that relates models that are at the same level of abstraction. This CM is propagated throughout the whole MTC relating models always at the same level of abstraction. This reduces the complexity to manually specify the correspondence relationships at a high-level of abstraction and to automatically derive the correspondence relationships throughout the whole set of transformation steps. Additionally, we take advantage of the same level of abstraction at the lowest level in order to perform the compositions between homogeneous models and to check consistency between heterogeneous models.

We define correspondence relationships between models at the same level of abstraction in our case study. The answer to the question "*Is the new concern specified at the same level of abstraction as the elements it affects?*" is **YES**, we specify the correspondence relationship at the same level of abstraction. These correspondence relationships identify which elements are affected by new requirements at a high-level of abstraction.

7.3.7 Criterion 7 (C7): Common integration mechanism

We want to use a reusable integration mechanism that allows to integrate the existing models with the new concern-specific requirements. Using a reusable

integration mechanism will reduce the costs of adding different concern-specific requirements.

We transform each concern-specific model (i.e., Business model, Security model, Navigation model) in our case study into platform-specific models that conform the same metamodel (i.e., Java metamodel). As previously explained, having each model conform to the same metamodel allows to perform a homogeneous composition and to reuse the composition mechanism. In our case we have a single composition mechanism that is used to compose three models where different concerns are specified in terms of Java concepts.

In our case study, we use a composition mechanism to integrate multiple models. The answer to the question "*Is it possible to use a common integration mechanism?*" is **YES**, we use the same composition mechanism for integrating models where business, security and navigation were specified independently.

7.4 Research Goals

After analyzing our approach using the key criteria that we presented in Section 3.3.1 we can now check if the research goals that we presented in Section 1.2 are fulfilled by our approach. We chose these goals in order to avoid the problems that we presented in Section 1.1.

7.4.1 General Goal: Non-invasive evolution of an MTC

Our general research goal was to avoid the modification of the original MTC. In other words, the main purpose of our research was to preserve the original metamodels, models and transformations once the new concern was introduced. As our case study shows, we are capable of adding three new concern-specific MTCs (i.e., Security MTC, Navigation MTC and Presentation MTC) without performing any change to the original Business MTC. This avoids the maintainability and understandability detriment of the original Business MTC assets. Furthermore, we kept conformance relationship between models and metamodels because the metamodel remains unchanged. Similarly, the original transformations remain compatible with the MTC metamodels. Moreover, because no artifact is modified, the *ripple effect* problem did not arise. Therefore, we can say that the original Business MTC remained oblivious of the new added MTCs.

7.4.2 Goal 1 (G1): Concern-specific modularization

By creating concern-specific MTCs (i.e., Security MTC, Navigation MTC and Presentation MTC), we achieve our first specific goal, which was to encapsulate the introduced changes in concern-specific modules. In our approach

we group each concern-specific artifact in a concern-specific MTC. This allows the *MTC developer* to focus on each concern-specific metamodel or transformation when he is building the MTC. In addition, the *application modeler* can independently specify the concern-specific requirements using suitable DSML concepts. Finally, we keep the original assets unmodified, thus improving the maintainability and understandability of the MTC by following the SoC principle. This goal is achieved because our approach obtains positive grades in *Criterion 1*, *Criterion 3*, *Criterion 4*, and *Criterion 7*.

7.4.3 Goal 2 (G2): Specifying the different concerns at a high-level of abstraction

The second specific goal is achieved by using concern-specific metamodels (i.e., Business metamodel, Security metamodel, Navigation metamodel, Presentation metamodel) that allow the specification of each concern-specific requirement at a high-level of abstraction. As explained previously, the use of a high-level metamodel gives the domain and concern experts the appropriate abstractions to define the application specification. Additionally, this goal is achieved because our approach obtains positive grades in *Criterion 1*, *Criterion 2* and *Criterion 3*.

7.4.4 Goal 3 (G3): Enabling an oblivious mechanism to integrate new concern-specific requirements

Finally, we use an oblivious composition mechanism that facilitates the integration of the three Java models as well as the consistency check of the composed model with the generated JSF model. The composition mechanism is generated from the correspondence derivation model by a HOT that allows having a metamodel independent composition mechanism that can be reused. Moreover, we choose to perform the composition at the lowest level of abstraction where we can have models that conform to the same metamodel. Therefore, this goal is achieved because our approach obtains positive grades in *Criterion 5*, *Criterion 6* and *Criterion 7*.

7.5 Limitations

One of the limitations of our approach has is the concern model co-evolution. This problem occurs when one of the high-level model needs to evolve and the other models will be required to be evolved as well. This is a typical problem in AOP when the base program is modified, and the pointcut expressions become inconsistent with it [KS04].

For instance, if the business model is modified, then all the models that have correspondence relationships with it (i.e., security and navigation models) will be required to be adapted in order to become consistent with the changes. Additionally, the presentation model that is related with the navigation model will be required to be changed as well. In spite of the above, the correspondence relationships between the models will help the *application modeler* to identify which elements need to be adapted to become consistent with the changes.

An additional current limitation is that only two concern-specific models can be related by a CM. However, a typical situation arises when several concerns are specified and interact between each other. This is a well-know problem in the AOSD community, and it affects our approach as well. In the current state of our work there is no possibility to express how more than two concern-specific models interact between each other or in which order they need to be integrated.

7.6 Summary

Validation of our work was presented in this chapter. This validation was performed with the help of a case study where three concern-specific MTCs (i.e., Security MTC, Navigation MTC, and Presentation MTC) were added to an existing Business MTC. Subsequently, we included an analysis of our approach using the key criteria that were presented in Section 3.3.1. All these key criteria are fulfilled by our approach showing the good features of it in the evolution of an existing MTC. Next, these key criteria are used to verify how our approach achieve the goals presented in Section 1.2. Finally, we discuss the limitations of our work in Section 7.5.

Chapter 8

Conclusion

8.1 Introduction

In this dissertation, we proposed a novel strategy to evolve an existing Model Transformation Chain. We have shown that our approach helps to add new concern-specific MTCs that can be combined with the existing one.

In this chapter we summarize the work presented in this dissertation and situate our work in the context of the research goals that were presented in the introduction. Next, we summarize the contributions made and explore alternatives for future research.

8.2 Summary

A large number of *Model-Driven Engineering* (MDE) implementations promote the use of models expressed in terms of problem domain concepts (e.g. Bank Account, Insurance Claim) as the prime artifact to develop software. These models, to which we refer as high-level models, are used as input for a *Model Transformation Chain* (MTC). This chain is a sequence of transformation steps that converts the high-level model, which is rooted in the problem domain, into a low-level model that is rooted in the solution domain (e.g., Java, C#). In addition to the translation from problem domain concepts to solution domain concepts (e.g., Java Class, Java Annotation), the transformation chain adds implementation details at each transformation step. Finally, the latest step in the chain is a model-to-text transformation that produces the code of the software system.

Evolution is an inherent characteristic of software systems. For instance, a software system must evolve if it is required to include new functionality, new non-functional properties, or the migration of the technology platform. Similar to this, MTCs are also susceptible to evolution. The evolution of an

MTC confronts us with several problems, mainly related to the strong dependencies between metamodels and models, metamodels and transformations, and between each transformation step and those which follow it.

The particular problem we address in this dissertation is the addition of a new concern (e.g., security, monitoring, business rules, etc.) that was not anticipated in the existing MTC implementation. No real problem arises if the new concern can be cleanly expressed using the existing high-level metamodel. However, if it is not the case, to extend the existing high-level metamodel with new concepts, such as security concepts into a business domain metamodel, arises some issues: 1) the existing metamodel has to be polluted with concepts that do not belong to the main problem domain, 2) if the metamodel is adapted, it is possible that the conformance relationship between metamodels and models is broken 3) including new elements in the application model produces a (large) monolithic model reducing maintainability, and 4) to manage the new concepts the transformation chain requires to adapt the existing steps or to add new ones. These changes increase the dependencies among the transformation chain steps adding complexity to it. These issues increase the difficulty to evolve the existing MDE implementation and to maintain applications.

We propose an approach that follows the SoC principle by adding new concern-specific MTCs. Our strategy consists of specifying the new concern in a separate high-level model, which leaves the original model untouched and oblivious of the added concern. This new model is specified using concepts close to the concern domain to keep our solution in line with the original vision of MDE. Therefore, we have two high-level models that conform to two different metamodels. In order to obtain the final application, it is necessary to compose both models, if the composition is executed at a high-level, we face a *heterogeneous composition* because both models conform to two different metamodels (e.g., composition of a business entity from the business domain and a secured resource from the security domain). A heterogeneous composition is a complex task and requires a particular composition mechanism for every added concern. Instead, we align the high-level models using a *Correspondence Model* (CM) [BBDF+06]. A CM is a model that explicitly describes the relationships among the elements of different models. We use correspondence relationships to identify the elements to compose. We have developed a strategy to derive the correspondence relationships throughout the various steps of the transformation chain. We postpone the composition to the lowest-level avoiding the overhead of developing a composition mechanism for each additional concern. At the lowest-level, every model conforms to the same metamodel (e,g,. Java metamodel) or to metamodels that are extensions of this metamodel. Additionally, a derived low-level CM relates these models maintaining the relationships defined in the high-level CM. The low-level CM is derived by an automatic transformation that uses as inputs the high-level

CM, trace models, which relate the elements of the high-level models and their generated elements in the low-level models, and a set of constraints defined in the *Correspondence Derivation Model* (CDM).

Having models conform to the same low-level metamodel (e,g,. Java metamodel) and a low-level CM relating these models allows us to do a *homogeneous composition* (e.g., composition of two Classes). This reduces the complexity of the composition and it is possible to use a single composition mechanism for different concerns. Moreover, it is possible to go a step further and transform the models into code to take advantage of existing general-purpose composition mechanisms such as AOP. Finally, we can transparently use the existing model-to-text transformation to generate the application code from the composed model.

In this dissertation we presented an approach that reduces the complexity of evolving a model transformation chain. Our approach offers several advantages: 1) it facilitates the modeling of different concerns in separated models and close to the problem domain, 2) it offers an automatic correspondence derivation mechanism to identify the elements to compose in the low-level models based on relationships defined in the high-level, 3) it eases the use of a single composition mechanism at a low-level of abstraction, 4) it reuses the existing assets (metamodels, models and transformations), and 5) it modularizes the changes in a new set of metamodels, models and transformations.

8.3 Contributions

In this section, we summarize the major contributions of this thesis, that have been published in [YCDW09a, YCDW09b, YCDW10a, YCDW10b].

8.3.1 A novel strategy to perform a non-invasive evolution of model transformation chains

The main contribution of this dissertation is a novel strategy to perform a non-invasive evolution of an existing model transformation chain. This evolution is limited to the introduction of new concerns. The approach modularizes the changes by adding additional concern-specific Model Transformation Chains.

The new MTCs allow to describe new concern-specific requirements in a high-level model by using concepts that belong to the particular concern and can be transparently used by concern experts. Each concern-specific high-level model is related to the existing high-level model using correspondence relationships. These relationships help to identify which elements are affected by the new requirements. Next, the correspondence relationships are propagated through the whole set of transformations until the lowest level of abstraction. At this level, it is possible to perform a homogeneous composition. This kind

of composition allows us to reuse a composition mechanism or to go a step further and take advantage of a code-level composition mechanism.

Our strategy avoids changes to the original MTC, and encapsulates the new concern concepts and the new required transformations in a concern-specific transformation chain.

8.3.2 A mechanism to automatically derive low-level correspondence relationships

In Chapter 4 we explained our correspondence relationships derivation mechanism that allows to propagate a Correspondence Model throughout a set of transformation steps. Finally, a new Correspondence Model between the generated lowest-level models is obtained. This derivation mechanism intensively uses tracing models, which relate the high-level source elements to the generated low-level elements. Additionally, our derivation mechanism uses a Correspondence Derivation Model in order to constrain the possible correspondence relationships that can be generated between the elements in the low-level models.

8.3.3 An analysis of the strategies that can be used to evolve an MTC

In Chapter 3 we presented an analysis of several strategies that can be used to add new concern-specific requirements to an existing MTC. These strategies are extracted from a study that we did, seeking for inspiration on multiple previous works that allowed the specification of several concerns and automatically compose them to produce a full application. We translated the ideas of these works to the field of MTC evolution.

In the presented analysis, we compare the chosen strategies with key criteria that allow us select the most suitable one. The analysis of the possible strategies was presented in [YCDS08].

8.3.4 Tool support

In Chapter 6, we presented a set of proof-of-concept tools, called *MTC Framework Toolkit* to support the evolution of an existing MTC, and the generation of applications using an evolved MTC. These tools were developed having in mind the two principal roles involved with an MTC: the *MTC developer*, which is responsible of building the metamodels and transformations, and the application modeler, who is responsible of creating the models and using the MTCs in order to generate applications.

On the one hand, the main goal of the toolkit regarding the *MTC developer* is to support him to evolve an existing MTC. This means to assist the developer in alignment of the two MTC and in the generation of the required transformations rules to compose the generated models or to check the consistency of the generated models. The tools that were developed having in mind the *MTC developer* are: Correspondence Derivation Modeler, the Composition Generator and the Correspondence Checker Generator.

On the other hand, the main goals of the toolkit regarding the *application modeler* is to help him to define correspondence relationships between the high-level models, and to execute the different transformations that derive correspondences, and to compose, or to check models. The tools that were developed having in mind the *MTC developer* are: the Correspondence Modeler and the Traceability Processor.

Finally, we implemented an ATL-VM extension that allows us to automatically generate tracing models. The objective of this extension was presented in [YW09].

8.4 Discussion

Our approach offers a strategy to align and modularize concern-specific MTCs. The models that are generated by these concern-specific MTCs are composed or checked, with the model generated by the existing MTC. We achieved this by using correspondence relationships, to identify corresponding elements between models that represent multiple concerns of an application. We implement a correspondence derivation mechanism that allows to propagate the correspondence relationships through the whole set of transformations steps. Finally, we offer a mechanism that allows composing low-level models that conform to the same metamodel.

The key of our approach relies on the ability of generating tracing links between the elements that belong to the models that are inputs of an MTC and the elements that are generated by the MTC. Tracing models are a fundamental element in the MDE field and they are required in order to offer an advanced mechanism to automatically generate tracing information when transformations are applied. Additionally, it is essential to have advanced tracing constructs that allow querying and navigating through complex nets of tracing models.

The different tracing links allow us to identify the elements that are generated from a pair of corresponding elements. However, it is necessary to have a mechanism to identify the correct matches between the generated elements. We define a Correspondence Derivation Metamodel that allows the *MTC developer* to specify constraints that restrict or allow pairs of elements that can be related by a correspondence relationship. If the Correspondence Derivation

Model (CDM) is created only with information about the target metamodels, then the CDM is more robust. However, this only happens when the transformation rules generate elements that conform to different metaclasses from a single element. If the transformation rules produce elements that conform to the same metaclass from a single source element, then it will be necessary to use the information of the transformation rules. The problem with this method is that transformation rules are more prone to change than metamodels. Therefore, if a change introduced into a transformation rule, then it will be necessary to check the CDM.

The main goal of our approach is to support the *MTC developer* in adding a new concern-specific MTC where the *Application modeler* can specify a set of concern-specific requirements independently from the original MTC. Our approach and the toolkit provided will help them to perform their tasks and obtain an application with the new concern requirements in it. However, our approach is only useful to add new complex and complete concerns that can be encapsulated in a concern-specific MTC. Our approach is too heavy-weight to introduce small changes to the existing artifacts. In other words, if it is required to add a couple of metaclasses or attributes to the existing metamodels, or to extend them, it is better to use a different approach. For instance, to directly modify the existing MTC artifacts or to use a combination of the approaches presented in in Section 2.3.

Furthermore, using correspondence relationships to align MTCs is a powerful tool for interoperable MTCs. This allow us to have several MTCs that work together to produce an application and the correspondence relationships will allow to represent the dependencies between them.

8.5 Future Work

This section describes a number of directions for future research in this area, as well as a number of possible improvements to our tool support.

8.5.1 Future Research

Our work opened several research paths that are discussed in this section.

Concerns interaction

A common situation that takes place whenever several concerns are specified is that they interact with each other. This is a well-know problem in the AOSD community, and it affects our approach as well.

Although we implemented a case study that aligns 4 independent MTCs, these MTCs are totally orthogonal to each other. This means that there is

no interaction between the four generated models, and the composition can be cleanly performed between pairs of models. Thisecomposition can be performed because there is no element in the lowest level models that is linked by a correspondence relationship to two other elements in two different models.

In order to extend the scope of our approach to the interacting concerns, we need a mechanism to express composition order. This will permit to specify which pair of elements should be composed first and which pair next. Additionally, we need a mechanism to detect interaction conflicts between overlapping concerns. This mechanism should inform the *application modeler* of the possible conflicts, and offer a set of possible solutions.

High-level models co-evolution

One of the problems of our approach is the model co-evolution. For instance, if the high-level model evolves, this will cause the rest of the high-level models (i.e., security, navigation and presentation) and the correspondence models, to become inconsistent. We need a mechanism that allows us to maintain the different models consistent or at least informs us the possible inconsistencies.

A possible solution is offered by Ruiz-Gonzalez et al. in [RGKK+09] who proposes an approach that allow the synchronization between multiple models where a Web application is specified. This approach helps to propagate the changes that are applied to one model into the other using correspondence relationships. These correspondence relationships are used to synchronize the multiple models.

We envision a solution that uses extended types of correspondence relationships. The correspondence relationships must allow for interoperating different MTCs and additionally, they must permit the synchronization of the different models.

Transformation Chain Interoperability

A possible research direction is to investigate the transformation chain interoperability. The strategy used in this dissertation to evolve MTCs can be generalized in order to describe relationships between multiple MTCs. These relationships can be used to allow for the interoperability among them.

To build a single "almighty" MTC that would be in charge of every design, implementation and specific platform concern is a complex task. Instead, we can use several smaller MTCs that are easier to develop and maintain, because each of them is independently developed focusing on a specific concern. However, the MTCs must interoperate to produce complete applications; this inherently creates dependencies between them due to the fact that each MTC generates a part of the final low-level model. We can use our approach to track dependencies between the MTCs, which can be used to automati-

cally derive *correspondence relationships* between the final models generated by each MTC. Similar to the work presented in this dissertation, we can use these relationships to integrate the generated models.

We envision an interoperability language that allows defining relationships between multiple MTCs, which can be implemented by using multiple transformations languages. Our interoperability language will maintain aligned the multiple MTCs and will integrate the models produced by these MTCs.

8.5.2 MTC Framework Toolkit Improvements

This section discusses the limitations of the MTC Framework Toolkit and suggests how these limitations can be mitigated. Directions for future work will also be discussed.

UML Profiles support

In the current implementation of our correspondence derivation mechanism, we only support compatibility constraints that use the metaclass types and the rule name/output variable as filters. Sometimes, when the models are defined using UML Profiles, these constraints are not enough to identify the generated elements.

It is possible that the MTCs are developed using DSMLs that use UML as the core modeling language and UML Profiles as an extension to specify domain concepts. When this happens, the Correspondence Derivation Modeler is limited and is unable to use *stereotypes* as a mechanism to identify the metaclass type. It is only able to use the UML metamodel metaclasses to specify the compatibility constraints.

We want to extend the Correspondence Derivation Modeler to allow the specification of compatibility constraints not only using metaclass types, but profile stereotypes as well. In other words, we want to use the domain concepts expressed as an UML Profile as elements to permit or restrict the propagation of correspondence relationships.

OCL based compatibility constraints

Moreover, we need to provide an extension that allows the use of OCL constraints to improve the identification of the generated elements using metamodel concepts. This will reduce the use of rule name/output variable information due to its fragility.

Although using rule name/output variable information to extend the compatibility constraints helps to increase the ability of identifying generated elements, the transformation rules are more susceptible to change than metamodels. This means that if the transformation rules are modified, the CDM

must be checked to verify the consistency with the new version of the transformations.

A possible way to avoid this situation is to use metamodel-based compatibility constraints, which can be enriched with OCL expressions that extend the scope of the constraints. If we provide a mechanism to add complex OCL expressions to the compatibility constraints, we will reduce the probability of using rule name/output variable information to identify the generated elements.

Constraining the Correspondence Modeler

The current implementation of the Correspondence Modeler allows the definition of correspondence relationships between elements that conform any metaclass. This represents a usability problem with big models due to the huge amount of possible relationships. In Section 5.2.1 we present an extension to the Correspondence Metamodel that allows to constrain the possible correspondence relationships that can be created. This means that it is possible to define a pair of metaclasses that can be used to define valid relationships. However, this extension will require modifying the Correspondence Modeler.

We have in mind an extension that allows loading text based OCL constraints that specify the pairs of valid metaclasses. With this type of extension the *MTC developer* can pre-configure the Correspondence Modeler for a pair of metamodels, and the *application modeler* will have a restricted modeler that only allows the definition of constraints between the allowed metaclasses.

Correspondence derivation performance

The Correspondence Derivation Transformation (CDT) is a declarative ATL rule. Although declarative style is the ATL preferred style, it has poor performance when several models are used as inputs of a transformation. Therefore, the generated CDT presents a poor performance when a correspondence model is derived from models with a large number of elements. A possible solution is to use the ATL imperative style to improve the performance of the CDT.

An interesting alternative to improve performance in the correspondence relationship resolution is the use of a *solver*. The use of solvers will help us to obtain the low-level correspondence model. In order to use solvers we will need to translate the high-level correspondence model, the two trancing models, the correspondence derivation model, and the input and output elements into facts that the solver can understand and resolve. Additionally, the use of solvers will help us to express more complex derivation constraints.

Advanced traceability operations

To have a complete traceability solution for the MTC Framework Toolkit it is necessary to consider several possible improvements. First, in order to calculate transitive closures over a set of tracing models, the current implementation composes the different models and generates a full model. This leads to a loss of information of the intermediate traces. We envision a solution that allows us to use traceability dedicated operations to navigate through the different tracing steps involved in a transformation chain ,without actually composing the trace models. Second, the current implementation of ATL does not trace the elements created by a *called rule*. It is necessary to define a representation for these elements, and relate them with the matched rules that trigger the called rule. Finally, in some cases the rule's name and the variable's name is not the most suitable meta-data for the tracing links and a more customized information is required. This could be possible if we add *trace annotations* to the transformation rules that can be added to the trace model automatically.

Appendix A

Metamodels

In this appendix the metamodels used in this example are briefly presented.

A.1 Business Metamodel

The goal of the Business metamodel is to specify Enterprise Applications using high-level business concepts that are free of technological platform details. The focus of this metamodel is to describe the business, the entities that belong to it, the relationships between them, and the services that each one offers. Figure A.1 presents the metamodel. The main concepts of this metamodel are:

Business

An instance of the *Business* metaclass represents the business of the Enterprise Application that is described in the model (e.g., project management, risk management). This metaclass is the root of the model.

BusinessEntity

The *BusinessEntity* metaclass is the main concept of a business model. This metaclass represents all the entities that belong to the business (e.g., project, risk). The generated application will manage the information of all the modeled entities.

Attribute

The metaclass *Attribute* represents the attributes of a *BusinessEntity* (e.g., the name of a project).

Association

The metaclass *Association* represents the relationships among the different *BusinessEntities* (e.g., the relationship between a project and the risks that threaten it).

Service

The metaclass *Service* represents the services that each *BusinessEntity* offers (e.g., the services that allow to add new risk to a project). This metaclass is specialized in the CRUD services that each entity will offer. The main specialization are: *Create*, *Detail*, *List*, *Update*, *Delete*.

A.2 Architecture Metamodel

The main objective of the *Architecture Metamodel* is to provide concepts that allow the description of a multi-layered application. Additionally, this metamodel offers some platform independent architectural patterns. Figure A.2 presents the metamodel. The main concepts of this metamodel are:

Layer

Layer is the main concept is *Layer* of the Architecture Metamodel. This is an abstract concept specialized into the concrete architectural layer. The specializations are: *PresentationLayer, ApplicationLayer, SystemServicesLayer, BusinessServicesLayer, PersistenceLayer* and *DataSourceLayer*.

Communication

The *Communication* metaclass represents a communication channel between two layers. This metaclass is constrained to to only communicate "adjacent" layers. The constraints are expressed as OCL constraints in the metamodel.

BusinessDelegate

The *BusinessDelegate* metaclass represents an architectural pattern that allows to encapsulate the business services of an application.

BusinessObject

The *BusinessObject* metaclass represents an architectural pattern that allows to transport the data of an entity between different layers.

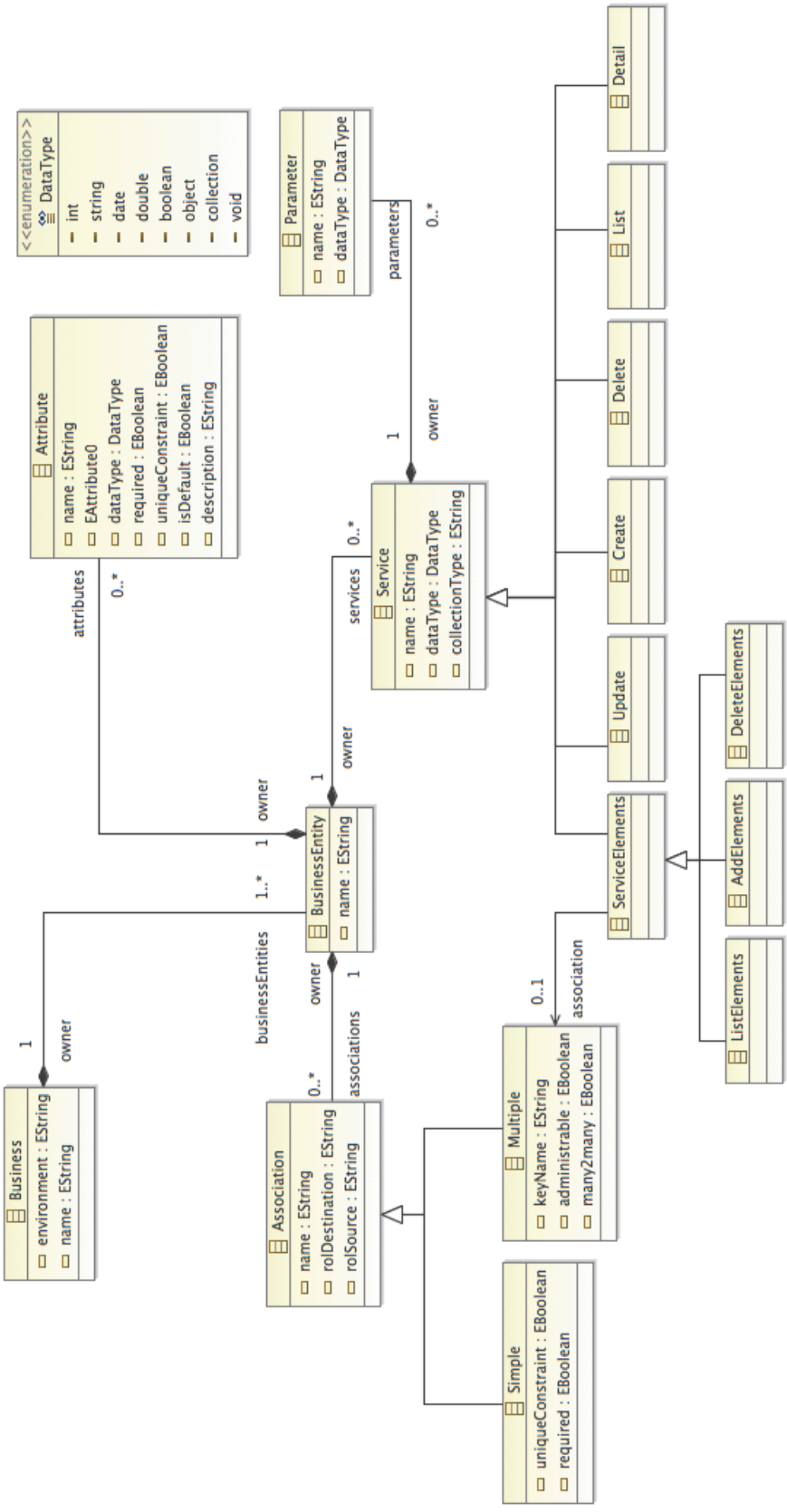


Figure A.1: Business Metamodel

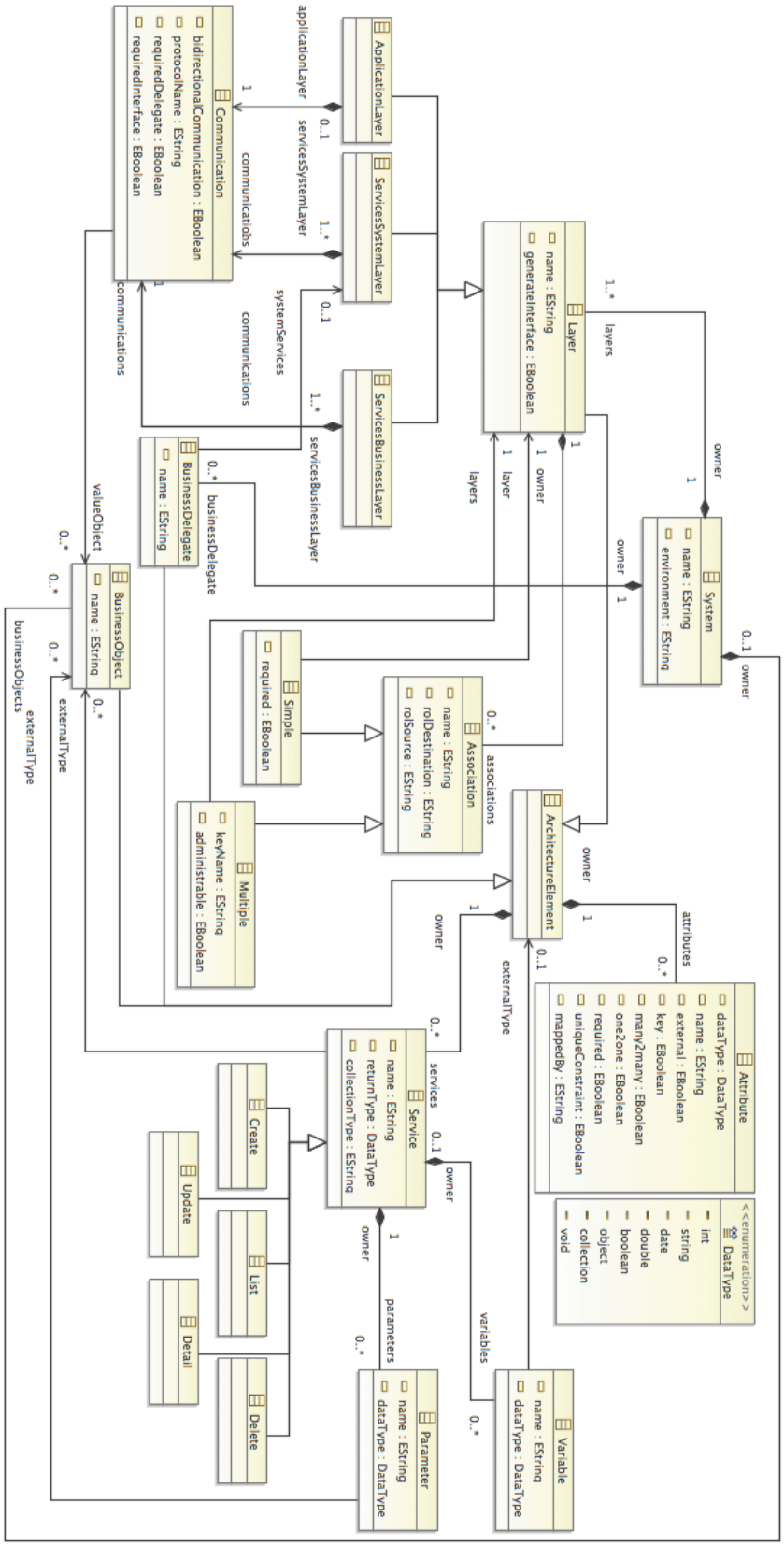


Figure A.2: Architecture Metamodel

A.3 JEE Metamodel

The main objective of the *Java Enterprise Edition Metamodel* is to provide concepts that belong to the JEE Application Server platform. Figure A.3 presents the metamodel. This metamodel contains platform specific concepts of JEE such as:

EntityBean

An instance of the *EntityBean* metaclass will represent the persistent data of a business entity. A *BusinessServicesLayer* instance in the Architectural model will be transformed into an instance of the *EntityBean* metaclass.

SessionBean

An instance of the *SessionBean* metaclass will provide the services that access or modify the information of a business entity. A *SystemServicesLayer* instance in the Architectural model will be transformed into an instance of the *SessionBean* metaclass.

ConnectionManager

The *ConnectionManager* metaclass represent a data access manager. An instance of this metaclass will manage all the connections to the database engine.

EJBDeploy

The *EJBDeploy* metaclass represents a JEE deployment unit. An instance of this metaclass will contain all the elements of an application module and will be executed in a JEE Application Server.

A.4 Java Metamodel

This appendix a summary of the MoDisco Java metamodel description presented at http://wiki.eclipse.org/MoDisco/J2SE5).

The MoDisco Java metamodel is the reflection of the Java language, as defined in version 3 of "Java Language Specification" from Sun Microsystems ("JLS3" corresponds to JDK 5). The Java metamodel contains 126 metaclasses. To better understand it, we will introduce its main features (metaclasses and links).

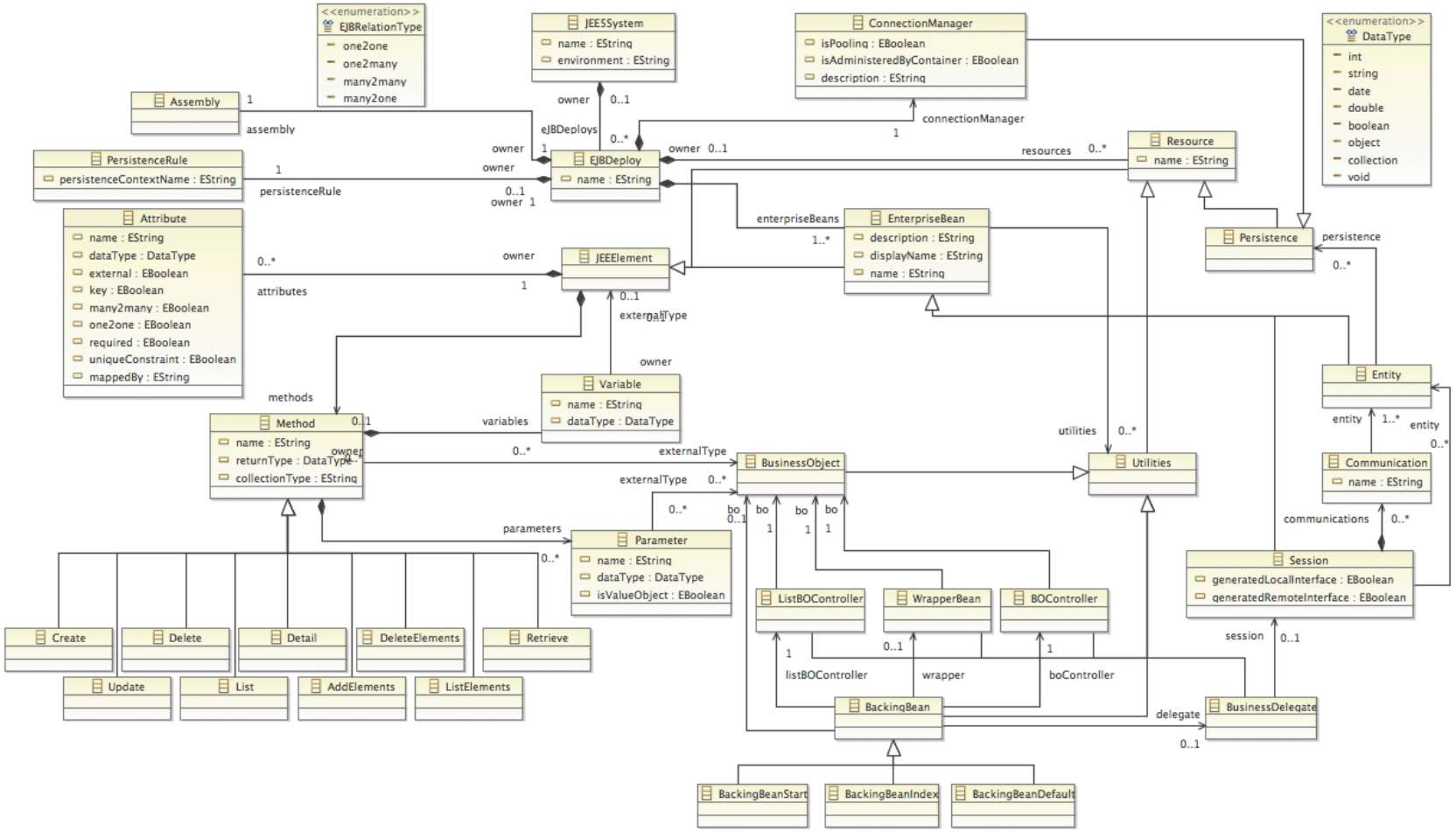


Figure A.3: JEE Metamodel

ASTNode

Every metaclass (apart from the Model metaclass) inherits from *ASTNode*. As its name indicates, *ASTNode* represents a graph node. *ASTNode* has a reference to the *Comment* metaclass because almost every java element can be associated to a comment (block or line comment and Javadoc).

Model, Package, & AbstractTypeDeclaration

The root element of each Java model is an instance of the *Model* metaclass. It is a translation of the concept of java application, so it contains package declarations (instances of the *Package* metaclass). And package declarations contain type declarations (instances compatible with the *AbstractTypeDeclaration* metaclass), and so on.

NamedElement & notion of Proxy

A lot of java elements are `named`, and this name could be considered as an identifier: methods, packages, types, variables, fields, etc. So all the corresponding metaclasses inherit from the *NamedElement* metaclass.

Another goal of this metaclass is to indicate whether an element is part of the current Java application or not (element from an external library of from the JDK). So, external elements are tagged as `proxy` through a dedicated attribute and can be easily filtered. For example, instruction `System.out.println` has been decomposed into three named elements (one class, one variable and one method) the definitions of which are not part of the current Java application. So, the attribute `proxy` of these elements has been initialized to true.

TypeAccess, PackageAccess, SingleVariableAccess, UnresolvedItemAccess

To represent links between Java elements, metaclasses *TypeAccess*, *PackageAccess*, *SingleVariableAccess* and *UnresolvedItemAccess* were introduced. Each allows to navigate directly to a *NamedElement* (proxy or not) in the graph.

A `TypeAccess` represents a reference on a type. A `PackageAccess` represents a reference on a package. A `SingleVariableAccess` represents a reference on a variable. On the contrary, references to methods are direct.

BodyDeclaration

A type declaration has different kinds of contents : fields, methods, static block, initialization block or other type declarations. All of these elements are of type BodyDeclaration metaclass.

For more information, please access MoDisco online information at `http://wiki.eclipse.org/MoDisco/J2SE5`.

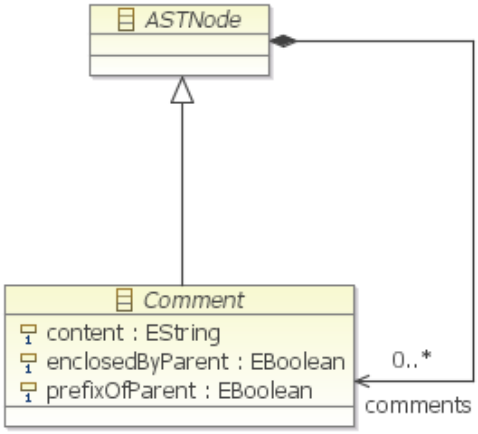


Figure A.4: ASTNode metaclass

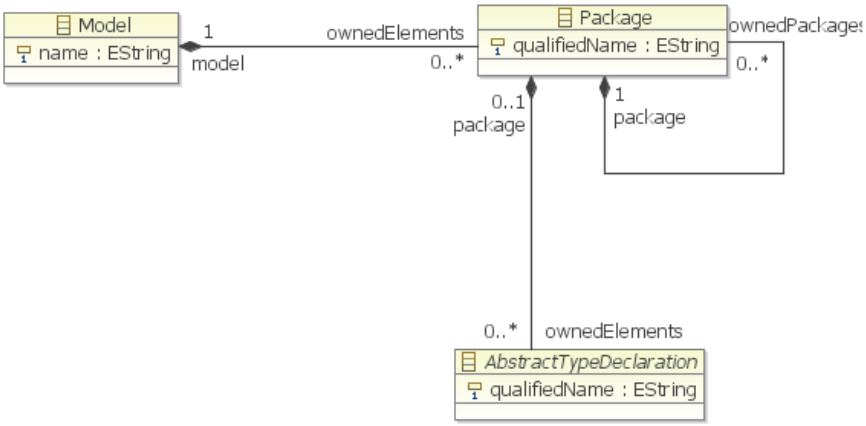


Figure A.5: Model, Package & type declaration superclass

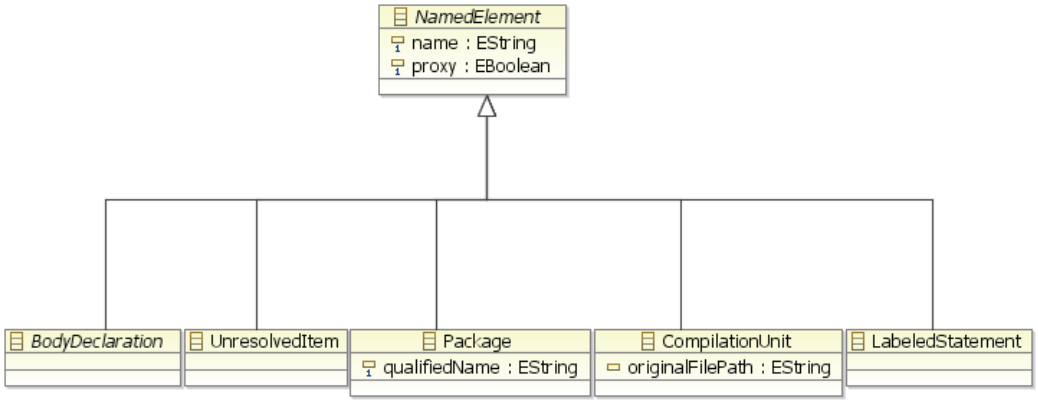


Figure A.6: NamedElement and its hierarchy

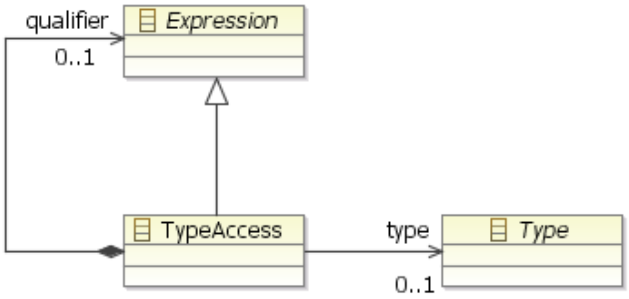


Figure A.7: TypeAccess

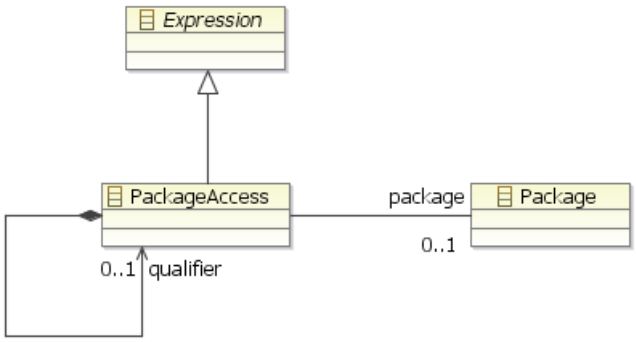


Figure A.8: PackageAccess

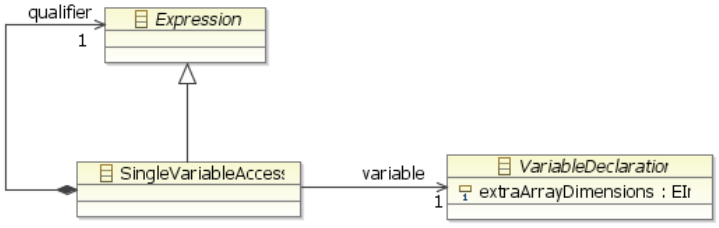


Figure A.9: SingleVariableAccess

Figure A.10: MethodInvocation

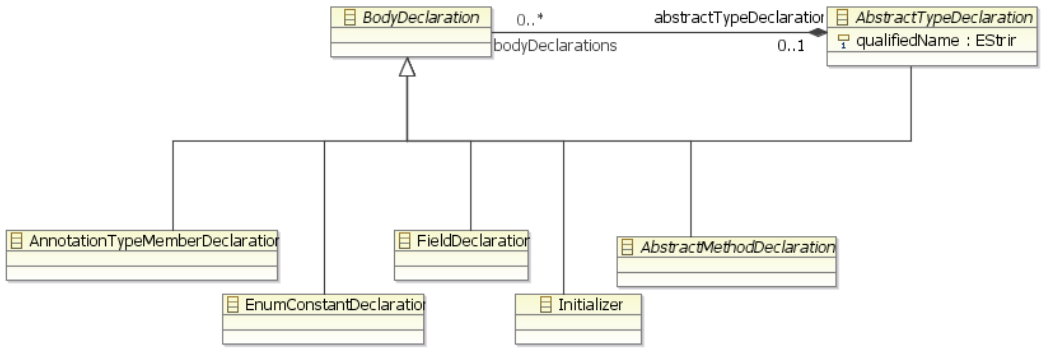


Figure A.11: BodyDeclaration and its hierarchy

A.5 Security Metamodel

The Security metamodel used in this research is based on the *SecureUML* metamodel presented in [LBD02], which in turn is based on the *Role Based Access Control* (RBAC) model [SCFY96]. The goal of this metamodel is represent authorization policies based of the RBAC model. Figure A.12 presents the metamodel. The main concepts of this metamodel are:

Resource

An instance of the metaclass *Resource* is an element that we need to protect. This is an abstract concepts that needs to be extended. In our case study we extended it into *ResourceEntity*, *ResourceAttribute*, and *ResourceOperation*.

Role

The *Role* metaclass is the main concept to assign access permissions over the application *Resources*. A *Role* can be composed of *Groups* and *Users*.

Action

The metaclass *Action* represent the type of operations that can be performed over a *Resource*. This is an abstract metaclass that needs to be extended. In our case study we extended it for each kind of resource.

Permission

The *Permission* metaclass combine several actions over the *Resources*, and assign them to the *Roles*.

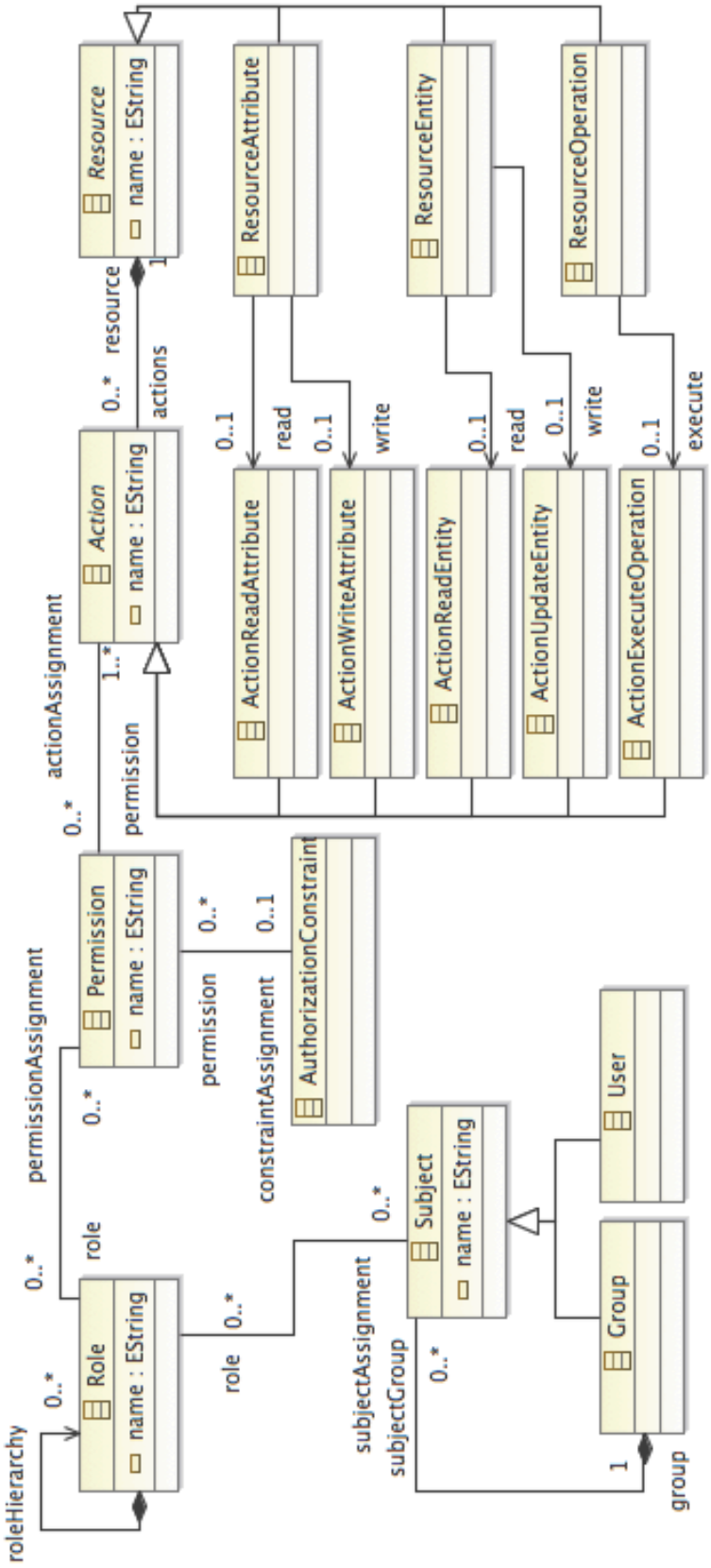


Figure A.12: Security Metamodel

A.6 Navigation Metamodel

The *Navigation metamodel* is defined using platform independent Web navigation concepts. This metamodel contains concepts that allow the representation of multiple navigations paths of a Web application. Figure A.13 presents the metamodel. The main concepts of this metamodel are:

NavigationFlow

The metaclass *NavigationFlow* contains a set of navigation nodes that are connected by navigation links allowing the final user to interact and navigate between the different application services.

NavigationClass

A *NavigationClass* represents a navigation node that allow the retrieving or capturing of information.

ProcessClass

A *ProcessClass* represents a navigation node where information is processed by the application.

Link

A *Link* represents a navigation path that interrelates the two navigation nodes. Each navigation node can have multiple links to other nodes or to other navigation flow.

A.7 Presentation Metamodel

The *Presentation metamodel* is defined using platform independent Web presentation concepts. This metamodel contains concepts that allow to represent the different Web pages of a Web application and their respective content. Figure A.14 presents the metamodel. The main concepts of this metamodel are:

Page

The main concept of the Presentation metamodel is the metaclass *Page*. This metaclass represents a Web page.

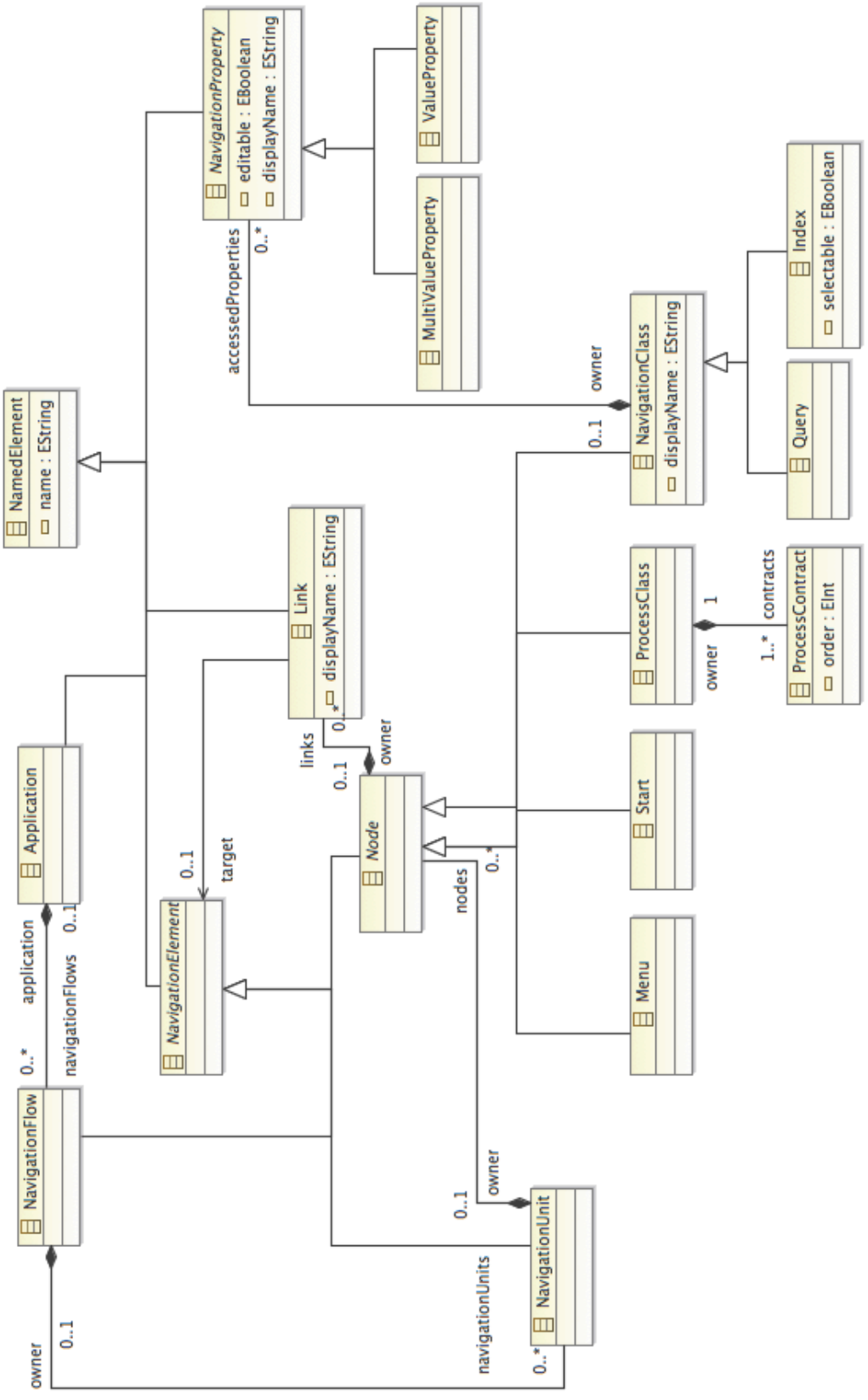


Figure A.13: Navigation Metamodel

View

A *Page* can have multiple *Views*. This metaclass represent the multiple simultaneous views of a page.

PresentationClass

Each *View* can be associated with a *PresentationClass*. A *PresentationClass* is an element that represents content that is provided by the application logic.

UIElement

A *PresentationClass* contains a set of *UIElements* that represent the Web user interaction components that displays and captures information. The *UIElements* are specialized in several metaclasses, such as: *TextInputs*, *Tables*, *Buttons*, etc.

A.8 JSF Metamodel

The *JSF Metamodel* is defined using the concepts of the technological platform *JavaServer Faces* (JSF). This metamodel contains concepts that allow the representation the different JSF pages and their user interface components. For instance, to present information, to capture data and to perform advanced interaction with the end-user. Figure A.15 presents the metamodel. The main concepts of this metamodel are:

Page

The main concept of the JSF metamodel is the metaclass *Page*. This metaclass represents a JSF page.

Folder

A *Page* is contained in a *Folder*.

UIElements

UIElements are the user interface components. There are several types of components such as: *Forms*, *TextFields*, *Tables*, *Buttons*, etc.

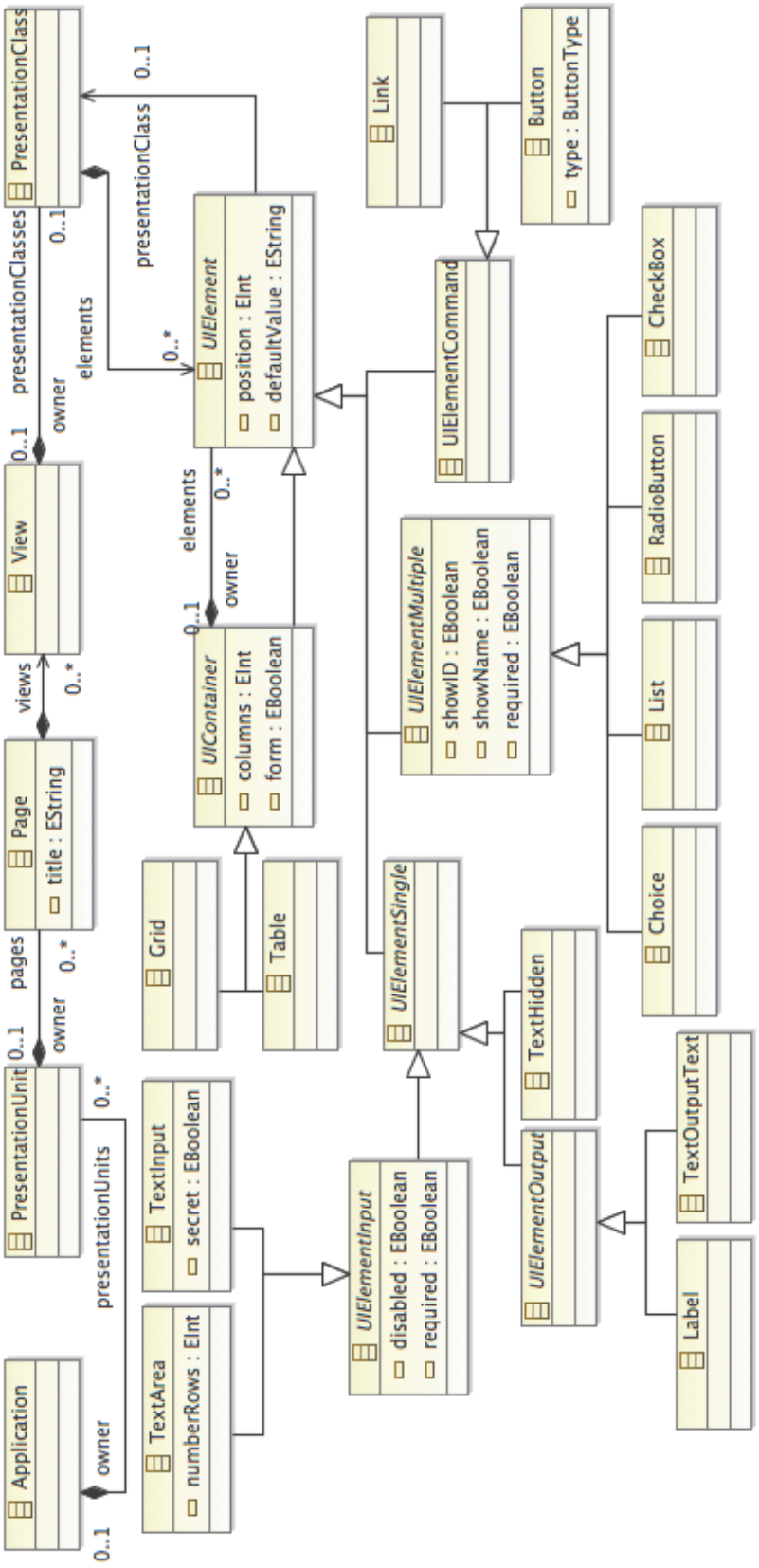


Figure A.14: Presentation Metamodel

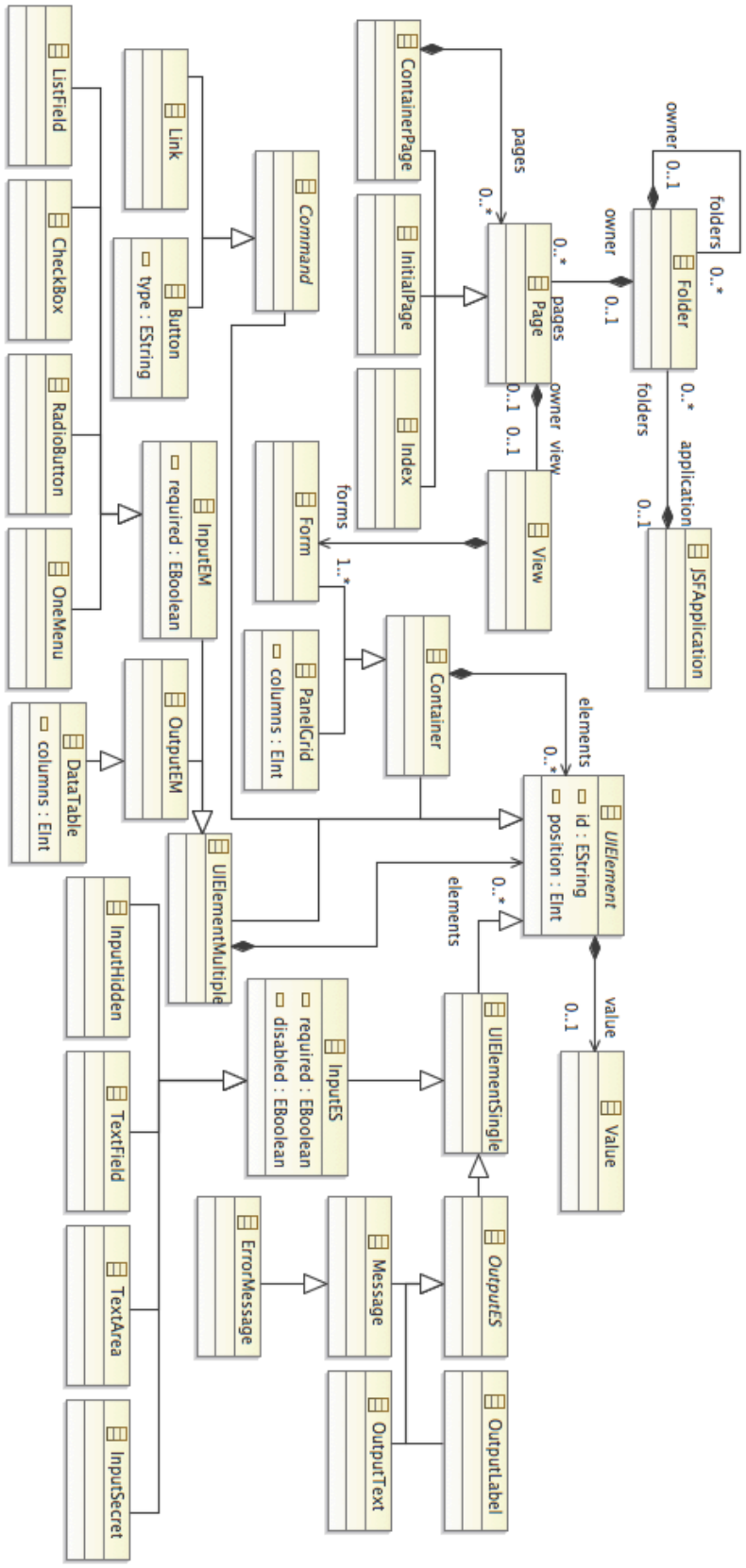


Figure A.15: JSF Metamodel

Appendix B

Transformation Rules

B.1 Tracing models Composer and Verifier Transformation

```
-------------------------------------------------------------
--  TRACE MERGE TRANSFORMATION
--    SOURCE METAMODEL: TRACE : Tracing Metamodel
--    TARGET METAMODEL: TRACE : Tracing Metamodel
-------------------------------------------------------------

module TraceMerge;

create outTrace : TRACE from inTrace : TRACE, inTrace2: TRACE, outM : MM;

-- ===========================================================================
-- helper attributes begin
-- ===========================================================================

helper def : startLinks :  Sequence(TRACE!TransientLink) = TRACE!TransientLink
    .allInstancesFrom('inTrace')->flatten();

helper def : numOutElements : Integer =  TRACE!"ecore::EObject".
    allInstancesFrom('outM').size();

helper def : endLinks :  Sequence(TRACE!TransientLink) =TRACE!TransientLink.
    allInstancesFrom('inTrace2')->flatten();

helper def : getTargets (inElement : TRACE!TransientElement): Sequence(TRACE!
    TransientElement) =
  thisModule.endLinks->select(e |
    (e.getSourceElements()->select(f |
      (f.value = inElement.value)))->notEmpty())->flatten()->collect(e | e.
          getTargetElements());

-- ===========================================================================
-- helper attributes end
-- ===========================================================================

-- ===========================================================================
-- transformation rules begin
-- ===========================================================================
```

```
entrypoint rule createTransientLinkSet() {
  to
    trace : TRACE!TransientLinkSet (
      links <- thisModule.startLinks->collect(e |
        thisModule.getTraceLink(e))->flatten()
    )
  do {
    trace.links->collect(e | e.getSourceElements())->flatten()->size().debug('
        in');
    trace.links->collect(e | e.getTargetElements())->flatten()->size().debug('
        out');
    thisModule.numOutElements->debug('Real Output');
    ((trace.links->collect(e | e.getTargetElements())->flatten()->size()/
        thisModule.numOutElements)*100).debug('Trace Coverage');

  }
}

rule getTraceLink(inLink : TRACE!TransientLink) {
  to
    trace : TRACE!TransientLink (
      --rule <- inSource.getRule().toString(),
      sourceElements <- inLink.getSourceElements()->collect(e | thisModule.
          getSourceElement(e)),
      targetElements <- inLink.getTargetElements()->collect(e | thisModule.
          getTargets(e))->flatten()->collect(e | thisModule.getTargetElement(e
          ))
    )
  do {
      trace;
    }
}

rule getSourceElement(inElement : TRACE!TransientElement) {
  to
    outelement : TRACE!TransientElement (
      name  <- inElement.name,
      value <- inElement.value
    )
  do {
    outelement;
  }
}

rule getTargetElement(inElement : TRACE!TransientElement) {
  to
    outelement : TRACE!TransientElement (
      name  <- inElement.name,
      value <- inElement.value
    )
  do {
    outelement;
  }

-- ===================================================================
-- transformation rules end
-- ===================================================================
```

B.2 Correspondence Derivation Transformation

```
-------------------------------------------------------------------
--  CORRESPONDENCE DERIVATION TRANSFORMATION FOR BUSINESS/NAVIGATION
```

```
--    SOURCE MODEL cmIN : High-Level Correspondence Model
--    SOURCE METAMODEL: CM : Correspondence Metamodel
--    SOURCE MODEL leftIN : left high-level Model
--    SOURCE METAMODEL: LEFTIN : left high-level Metamodel
--    SOURCE MODEL leftOUT : left low-level Model
--    SOURCE METAMODEL: LEFTOUT : left low-level Metamodel
--    SOURCE MODEL rightIN : right high-level Model
--    SOURCE METAMODEL: RIGHTIN : right high-level Metamodel
--    SOURCE MODEL rightOUT : right low-level Model
--    SOURCE METAMODEL: RIGHTOUT : right low-level Metamodel
--    SOURCE MODEL leftTrace : left Trace Model
--    SOURCE MODEL righttTrace : right Trace Model
--    SOURCE METAMODEL: TRACE : TRACE Metamodel

--    TARGET MODEL: cmOUT : Low-level Correspondence Model
--    TARGET METAMODEL: CM : Correspondence Metamodel
-------------------------------------------------------------------

module EA_NAV_2_J2SE_J2SENAV_Derivation;

create cmOUT : CM from cmIN : CM, leftIN : LEFTIN, leftOUT: LEFTOUT, leftTrace
    :TRACE, rightIN : RIGHTIN, rightOUT : RIGHTOUT, rightTrace : TRACE;

-- ================================================================
-- helper attributes begin
-- ================================================================

helper def: cmModel : CM!AlignmentModel =
  OclUndefined;

helper def: cmLinks : Sequence(CM!AlignmentLink) = CM!AlignmentLink.
    allInstancesFrom('cmIN');

helper context CM!AlignmentLink def: isCorresponding (left : CM!EObject, right
     : CM!EObject) : Boolean =
  (thisModule.getTrace(left, 'leftTrace')->select(e | e.sourceElements->first
      ().value = self.left.ref)).notEmpty()
    and (thisModule.getTrace(right, 'rightTrace')->select(e | e.sourceElements
        ->first().value = self.right.ref)).notEmpty();

helper context CM!AlignmentLink def: isCorrespondingFiltered (left : CM!
    EObject, right : CM!EObject, rules : Sequence(TupleType(leftRule : String,
     rightTuple : String))) : Boolean =
  rules->select(r |
  (thisModule.getTrace(left, 'leftTrace')->select(e | r.leftRule = e.getName(
      left) and e.sourceElements->first().value = self.left.ref)).notEmpty()
    and (thisModule.getTrace(right, 'rightTrace')->select(e | r.rightRule = e.
        getName(right) and e.sourceElements->first().value = self.right.ref)).
        notEmpty() ).notEmpty() ;

helper def: getTrace(element : CM!EObject, model : String) : TRACE!
    TransientLink =
  TRACE!TransientLink.allInstancesFrom(model)->select(e |
    e.targetElements->select(f | f.value = element).notEmpty()
  );

helper context TRACE!TransientLink def: getName(element : TRACE!
    TrasientElement) : String =
  self.targetElements->select(e | e.value = element)->first().name;

  -- ================================================================
  -- helper attributes end
  -- ================================================================

  -- ================================================================
```

```
  -- transformation rules begin
  -- ================================================================================

rule CorrespondenceModel2CorrespondenceModel {
  from
    inmodel   : CM!AlignmentModel
  to
    outmodel  : CM!AlignmentModel (
          name  <-  inmodel.name
      )
    do {
      thisModule.matchModel <- outmodel;
    }
}

rule EnhanceLink_MethodDeclaration_MethodDeclaration {
  from
    leftElement : LEFTOUT!Model,
    rightElement : RIGHTOUT!Model,
      inLink : CM!AlignmentLink(
      inLink.isCorresponding(leftElement, rightElement)
      )
  to
    outlink : CM!AlignmentLink (
      model <- thisModule.matchModel,
      elements <- Sequence{leftEnd, rightEnd},
      left <- leftEnd,
      right <- rightEnd
    ),
    leftEnd : CM!LeftElement (ref <- leftElement),
    rightEnd : CM!RightElement (ref <- rightElement)
  }

rule EnhanceLink_PackageDeclaration_PackageDeclaration {
  from
    leftElement : LEFTOUT!PackageDeclaration,
    rightElement : RIGHTOUT!PackageDeclaration,
      inLink : CM!AlignmentLink(
      inLink.isCorrespondingFiltered(leftElement, rightElement,
        Sequence {
          Tuple{leftRule = 'JEE5System2Model.co', rightRule = 'System2Model.co
              '},
          Tuple{leftRule = 'JEE5System2Model.edu', rightRule = 'System2Model.
              edu'},
          Tuple{leftRule = 'JEE5System2Model.uniandes', rightRule = '
              System2Model.uniandes'},
          Tuple{leftRule = 'JEE5System2Model.system', rightRule = '
              System2Model.system'},
          Tuple{leftRule = 'JEE5System2Model.bo', rightRule = 'System2Model.bo
              '},
          Tuple{leftRule = 'JEE5System2Model.delegate', rightRule = '
              System2Model.delegate'}
        }
      )
      )
  to
    outlink : CM!AlignmentLink (
      model <- thisModule.matchModel,
      elements <- Sequence{leftEnd, rightEnd},
      left <- leftEnd,
      right <- rightEnd
    ),
    leftEnd : CM!LeftElement (ref <- leftElement),
    rightEnd : CM!RightElement (ref <- rightElement)
  }
```

```
rule EnhanceLink_NamedElementRef_NamedElementRef {
  from
    leftElement : LEFTOUT!NamedElementRef,
    rightElement : RIGHTOUT!NamedElementRef,
      inLink : CM!AlignmentLink(
      inLink.isCorrespondingFiltered(leftElement, rightElement,
        Sequence {
          Tuple{leftRule = 'BO2ClassDeclaration.typeBO', rightRule = '
              BusinessObject.typeBO'},
          Tuple{leftRule = 'BO2ClassDeclaration.typeCollection', rightRule = '
              BusinessObject.typeCollection'}
        }
      )
      )
  to
    outlink : CM!AlignmentLink (
      model <- thisModule.matchModel,
      elements <- Sequence{leftEnd, rightEnd},
      left <- leftEnd,
      right <- rightEnd
    ),
    leftEnd : CM!LeftElement (ref <- leftElement),
    rightEnd : CM!RightElement (ref <- rightElement)
  }

rule EnhanceLink_ClassDeclaration_ClassDeclaration {
  from
    leftElement : LEFTOUT!ClassDeclaration,
    rightElement : RIGHTOUT!ClassDeclaration,
      inLink : CM!AlignmentLink(
      inLink.isCorrespondingFiltered(leftElement, rightElement,
        Sequence {
          Tuple{leftRule = 'Delegate2ClassDeclaration.delegate', rightRule = '
              BusinessDelegate.delegate'},
          Tuple{leftRule = 'BO2ClassDeclaration.b', rightRule = '
              BusinessObject.bo'}
        }
      )
      )
  to
    outlink : CM!AlignmentLink (
      model <- thisModule.matchModel,
      elements <- Sequence{leftEnd, rightEnd},
      left <- leftEnd,
      right <- rightEnd
    ),
    leftEnd : CM!LeftElement (ref <- leftElement),
    rightEnd : CM!RightElement (ref <- rightElement)
  }

rule EnhanceLink_ParameterizedType_ParameterizedType {
  from
    leftElement : LEFTOUT!ClassDeclaration,
    rightElement : RIGHTOUT!ClassDeclaration,
      inLink : CM!AlignmentLink(
      inLink.isCorrespondingFiltered(leftElement, rightElement,
        Sequence {
          Tuple{leftRule = 'BO2ClassDeclaration.parameterizedType', rightRule
              = 'BusinessObject.parameterizedType'}
        }
      )
      )
  to
    outlink : CM!AlignmentLink (
```

```
      model <- thisModule.matchModel,
      elements <- Sequence{leftEnd, rightEnd},
      left <- leftEnd,
      right <- rightEnd
    ),
    leftEnd : CM!LeftElement (ref <- leftElement),
    rightEnd : CM!RightElement (ref <- rightElement)
  }

-- ================================================================
-- transformation rules end
-- ================================================================
```

Appendix C

ATL Tutorial

This appendix is a brief ATL tutorial that is extracted from the ATL User Guide (http://wiki.eclipse.org/ATL/User_Guide).

As an answer to the OMG MOF/QVT RFP, ATL mainly focuses on the model to model transformations. Such model operations can be specified by means of ATL modules. An ATL module corresponds to a model to model transformation. This kind of ATL unit enables ATL developers to specify the way to produce a set of target models from a set of source models. Both source and target models of an ATL module must be "typed" by their respective metamodels. Moreover, an ATL module accepts a fixed number of models as input, and returns a fixed number of target models. As a consequence, an ATL module can not generate an unknown number of similar target models (e.g. models that conform to a same metamodel).

An ATL module defines a model to model transformation. It is composed of the following elements:

1. A header section that defines some attributes that are relative to the transformation module;
2. An optional import section that enables to import some existing ATL libraries;
3. A set of helpers that can be viewed as an ATL equivalent to Java methods;
4. A set of rules that defines the way target models are generated from source ones.

Helpers and rules do not belong to specific sections in an ATL transformation. They may be declared in any order with respect to certain conditions (see ATL Helpers section for further details). These four distinct element types are now detailed in the following subsections.

Header section

The header section defines the name of the transformation module and the name of the variables corresponding to the source and target models. It also encodes the execution mode of the module. The syntax for the header section is defined as follows:

```
module module_name;
create output_models [from|refining] input_models;
```

The keyword `module` introduces the name of the module. The target models declaration is introduced by the `create` keyword, whereas the source models are introduced either by the keyword `from` (in normal mode) or `refining` (in case of refining transformation). The declaration of a model, either a source input or a target one, must conform the scheme `model_name : metamodel_name`. It is possible to declare more than one input or output model by simply separating the declared models by a coma. The following ATL source code represents the header of the Book2Publication.atl file, e.g. the ATL header for the transformation from the Book metamodel to the Publication metamodel:

```
module Book2Publication;
create OUT : Publication from IN : Book;
```

Import section

The optional import section enables to declare which ATL libraries have to be imported. For instance, to import the strings library, one would write:

```
uses strings;
```

Helpers

ATL helpers can be viewed as the ATL equivalent to Java methods. They make it possible to define factorized ATL code that can be called from different points of an ATL transformation. An ATL helper is defined by the following elements:

- a `name` (which corresponds to the name of the method);
- a `context` type. The context type defines the context in which this attribute is defined (in the same way a method is defined in the context of given class in object-programming);
- a `return` value type. Note that, in ATL, each helper must have a return value;
- an ATL `expression` that represents the code of the ATL helper;

- an optional set of `parameters`, in which a parameter is identified by a couple (parameter name, parameter type).

This is, for instance, the case for a helper that just multiplies an integer value by two:

```
helper context Integer def : double() : Integer = self * 2;
```

The ATL language also makes it possible to define attributes. An attribute helper is a specific kind of helper that accepts no parameters, and that is defined either in the context of the ATL module or of a model element. Thus, the attribute version of the double helper defined above will be declared as follows:

```
helper context Integer def : double : Integer = self * 2;
```

Rules

In ATL, there exist three different kinds of rules that correspond to the two different programming modes provided by ATL (e.g. declarative and imperative programming): the *matched rules* (declarative programming), *the lazy rules*, and the *called rules* (imperative programming).

Matched rules: The matched rules constitute the core of an ATL declarative transformation since they make it possible to specify:

1) for which kinds of source elements target elements must be generated,

2) the way the generated target elements have to be initialized.

A matched rule is identified by its name. It matches a given type of source model element, and generates one or more kinds of target model elements. The rule specifies the way generated target model elements must be initialized from each matched source model element. A matched rule is introduced by the keyword `rule`. It is composed of two mandatory (the source and the target patterns) and two optional (the local variables and the imperative) sections. When defined, the local variable section is introduced by the keyword `using`. It enables to locally declare and initialize a number of local variables (that will only be visible in the scope of the current rule). The source pattern of a matched rule is defined after the keyword `from`. It enables to specify a model element variable that corresponds to the type of source elements the rule has to match. This type corresponds to an entity of a source metamodel of the transformation. This means that the rule will generate target elements for each source model element that conforms to this matching type. In many cases, the developer will be interested in matching only a subset of the source elements that conform to the matching type. This is simply achieved by specifying an optional condition (expressed as an ATL expression, see OCL Declarative Expressions section for further details) within the rule source pattern. By this

mean, the rule will only generate target elements for the source model elements that both conform to the matching type and verify the specified condition.

The target pattern of a matched rule is introduced by the keyword `to`. It aims to specify the elements to be generated when the source pattern of the rule is matched, and how these generated elements are initialized. Thus, the target pattern of a matched rule specifies a distinct target pattern element for each target model element the rule has to generate when its source pattern is matched. A target pattern element corresponds to a model element variable declaration associated with its corresponding set of initialization bindings. This model element variable declaration has to correspond to an entity of the target metamodels of the transformation.

Finally, the optional imperative section, introduced by the keyword `do`, makes it possible to specify some imperative code that will be executed after the initialization of the target elements generated by the rule. As an example, consider the following simple ATL matched rule between two metamodels, *MMAuthor* and *MMPerson:*

```
rule Author {
  from
    a : MMAuthor!Author
  to
    p : MMPerson!Person (
      name <- a.name,
      surname <- a.surname
    )
}
```

This rule, called Author, aims to transform *Author* source model elements (from the *MMAuthor* source model) to *Person* target model elements in the *MMPerson* target model. This rule only contains the mandatory source and target patterns. The source pattern defines no filter, which means that all *Author* classes of the source *MMAuthor* model will be matched by the rule. The rule target pattern contains a single simple target pattern element (called p). This target pattern element aims to allocate a *Person* class of the *MMPerson* target model for each source model element matched by the source pattern. The features of the generated model element are initialized with the corresponding features of the matched source model element. Note that a source model element of an ATL transformation should not be matched by more than one ATL matched rule. This implies the source pattern of matched rules to be designed carefully in order to respect this constraint. Moreover, an ATL matched rule can not generate ATL primitive type values.

Lazy rules: Lazy rules are like matched rules, but are only applied when called by another rule.

Called rules: The called rules provide ATL developers with convenient imperative programming facilities. Called rules can be seen as a particular

type of helpers: they have to be explicitly called to be executed and they can accept parameters. However, as opposed to helpers, called rules can generate target model elements as matched rules do. A called rule has to be called from an imperative code section, either from a match rule or another called rule.

As a matched rule, a called rule is introduced by the keyword `rule`. As matched rules, called rules may include an optional local variables section. However, since it does not have to match source model elements, a called rule does not include a source pattern. Moreover, its target pattern, which makes it possible to generate target model elements, is also optional. Note that, since the called rule does not match any source model element, the initialization of the target model elements that are generated by the target pattern has to be based on a combination of local variables, parameters and module attributes. The target pattern of a called rule is defined in the same way the target pattern of a matched rule is. It is also introduced by the keyword `to`. A called rule can also have an imperative section, which is similar to the ones that can be defined within matched rules. In order to illustrate the called rule structure, consider the following simple example:

```
rule NewPerson (na: String, s_na: String) {
  to
    p : MMPerson!Person (
      name <- na
    )
  do {
    p.surname <- s_na
  }
}
```

This called rule, named *NewPerson*, aims to generate *Person* target model elements. The rule accepts two parameters that correspond to the *name* and the *surname* of the *Person* model element that will be created by the rule execution. The rule has both a target pattern (called p) and an imperative code section. The target pattern allocates a *Person* class each time the rule is called, and initializes the name attribute of the allocated model element. The imperative code section is executed after the initialization of the allocated element. In this example, the imperative code sets the *surname* attribute of the generated *Person* model element to the value of the parameter *s_ na*.

Bibliography

[ARNRSG06] Neta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45:515–526, July 2006. 2.2.6, 4.2.2

[BBDF+06] Jean Bézivin, Salim Bouzitouna, Marcos Del Fabro, Marie-Pierre Gervais, Fréderic Jouault, Dimitrios Kolovos, Ivan Kurtev, and Richard Paige. A canonical scheme for model composition. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg, 2006. 10.1007/11787044_26. (document), 1.3, 2.4.2, 2.4.2, 8.2

[BCA+06] Elisa Baniassad, Paul C. Clements, Joao Ara?, Ana Moreira, Awais Rashid, and Bedir Tekinerdogan. Discovering early aspects. *IEEE Software*, 23:61–70, 2006. 2.4.1

[BDL06] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions in Software Engineering and Methodology*, 15:39–91, January 2006. 3.4.1

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society. 2.2.1

[BH08] Nelis Boucké and Tom Holvoet. View composition in multiagent architectures. *International Journal of Agent-Oriented Software Engineering*, 2:3–33, January 2008. 2.4.1

[BJRV05] J Bézivin, F Jouault, P Rosenthal, and P Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. 2005. 2.2

[BLS03] Don Batory, Jia Liu, and Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. *SIGSOFT Software Engineering Notes*, 28:48–57, September 2003. 5.4

[Bri05] Johan Brichau. *Integrative Composition of Program Generators*. PhD. Dissertation, 2005. 5.4

[BWH+08] Nelis Boucké, Danny Weyns, Rich Hilliard, Tom Holvoet, and Alexander Helleboogh. Characterizing relations between architectural views. In *Proceedings of the 2nd European conference on Software Architecture*, ECSA '08, pages 66–81, Berlin, Heidelberg, 2008. Springer-Verlag. 5.4

[CD06] María Cibrán and Maja D'Hondt. A slice of MDE with AOP: transforming high-level business rules to aspects. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 170–184. Springer Berlin / Heidelberg, 2006. 10.1007/11880240_13. 3.4.5, 3.5

[CDR08] Antonio Cicchetti and Davide Di Ruscio. Decoupling Web application concerns through weaving operations. *Science of Computer Programming*, 70:62–86, January 2008. 2.4.1, 2.4.2, 3.4.2

[CDREP] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Meta-model differences for supporting model co-evolution. *Proc. of the 2nd Int. Workshop on Model-Driven Software Evolution (MoDSE 2008), Athens (Greece)*. 1.1.2, 2.3, 2.3.1

[CS09] Roberto Chinnici and Bill Shannon. *Java^{TM} Platform, Enterprise Edition (Java EE) Specification, v6*. Sun Microsystems, 2009. 1.1, 3.2, 4.3.3, 7.2.4

[CSW05] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling - A Foundation for Language Driven Development*. 2005. 2.2.1, 2.2.2

[DDZ08] Jürgen Dingel, Zinovy Diskin, and A. Zito. Understanding and improving UML package merge. *Software and Systems Modeling*, 7(4):443–467, October 2008. 5.4

[DFBV06] Marcos Didonet, Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving models with the Eclipse AMW plugin. *Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006, 2006. (document), 2.4.2, 2.4.2, 2.8, 5.2

[DHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14:199–245, April 2005. 2.4.1

[Dij82] Edsger W Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982. 1.2.2, 2.4

[DRMM+10] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing next generation ADLs through MDE techniques. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 85–94, New York, NY, USA, 2010. ACM. 2.4.1

[EMLB10] Anne Etien, Alexis Muller, Thomas Legrand, and Xavier Blanc. Combining independent model transformations. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2237–2243, New York, NY, USA, 2010. ACM. 2.3.2

[Fav03] Jean-Marie Favre. Meta-model and model co-evolution within the 3D software space. *Evolution of Large-scale Industrial Software Applications (ELISA 2003)*, 3:98–109, 2003. 1.1.2, 2.3, 2.3.1

[Fav04] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. In *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004. 2.2.2

[FHN06] Jean-Remy Falleri, Marianne Huchard, and Clémentine Nebut. C.: Towards a traceability framework for model transformations in Kermeta. *In: ECMDA- Traceability Workshop*, 2006. 1

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. 2.4.1, 2.4.2

[FTD08] Johan Fabry, Éric Tanter, and Theo DHondt. KALA: kernel aspect language for advanced transactions. *Science of Computer Programming*, 71:165–180, May 2008. 3.4.4

[GEM10] GEMS. Generic Eclipse Modeling System (GEMS). http://www.eclipse.org/gmt/gems/, 2010. 2.2.3

[GJCB09] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag. 2.3, 2.3.1, 5.3

[GME10] GME. Generic Modeling Environment (GME). http://www.isis.vanderbilt.edu/Projects/gme/, 2010. 2.2.3

[GS09] Hassan Gomaa and Michael E. Shin. Separating application and security concerns in use case models. *Proceedings of the 15th workshop on Early aspects*, pages 1–6, 2009. 2.4.1

[HCW07] Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wąsowski. Guided Development with Multiple Domain-Specific Languages. *In ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems (MODELS 2007*, 2007. 2.4.1

[HO93] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notices*, 28:411–428, October 1993. 5.4

[JCC09] José Uetanabara Júnior, Valter Vieira Camargo, and Christina Von Flach Chavez. UML-AOF: a profile for modeling aspect-oriented frameworks. *Proceedings of the 13th workshop on Aspect-oriented modeling*, pages 1–6, 2009. 2.4.1

[JFB08] Cédric Jeanneret, Robert France, and Benoit Baudry. A reference process for model composition. In *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*, pages 1–6, New York, NY, USA, 2008. ACM. 2.4.2, 4.3.5

[JK06] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM. 1.1.1, 2.2.4, 2.2.4, 3, 6.2, 6.5

[Jou05] Frederic Jouault. Loosely coupled traceability for ATL. *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, 91, 2005. 2, 2, 6.5

[KGZ09] Jochen M. Küster, Thomas Gschwind, and Olaf Zimmermann. Incremental Development of Model Transformation Chains Using Automated Testing. *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 733–747, 2009. 2.2.5

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 285–294, 2007. 1.1.3, 2.3.3, 3.3.1

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming*, May 1997. 1.1.1, 1.3, 2.4.1, 3.1, 3.3, 3.3.1, 5.4

[KS04] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004. 7.5

[LBD02] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-based modeling language for model-driven security. *UML 2002 - The Unified Modeling Language : 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings*, pages 426–441, 2002. 3.4.1, 4.3.2, 7.2.2, A.5

[LH09] Henrik Lochmann and Anders Hessellund. An Integrated View on Modeling with Multiple Domain-Specific Languages. *Proceedings of the IASTED International Conference Software Engineering (SE 2009)*, pages 1–10, February 2009. 2.4.1, 3.1, 3.3

[Lud03] Jochen Ludewig. Models in software engineering – an introduction. *Software and Systems Modeling*, 2:5–14, 2003. 10.1007/s10270-003-0020-3. 2.2.1

[MBC09] Ana Luisa Medeiros, Thais Batista, and Christina Chavez. MARISA-DP – from architecture to design: an MDD approach.

Proceedings of the 15th workshop on Early aspects, pages 37–42, 2009. 2.4.1

[MDT07] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Journal Information and Software Technology*, 49:12–31, January 2007. 2.4.1

[Met10] MetaCase. Metaedit+. http://www.metacase.com/, 2010. 2.2.3

[MFB09] Pierre-Alain Muller, Frédéric Fondement, and Benoît Baudry. Modeling Modeling. *Model Driven Engineering Languages and Systems*, 5795:2–16, 2009. 2.2.1

[MM03] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. 2003. 2.2.1, 2.2.1

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70 –93, January 2000. 2.4.1

[MV02] Pieter J. Mosterman and Hans Vangheluwe. Guest editorial: Special issue on computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12:249–255, October 2002. 2.4.1

[MV09] Bert Meyers and Hans Vangheluwe. Evolution of Modelling Languages. *BENEVOL 2009 The 8 th BElgian-NEtherlands software eVOLution seminar*, page 43, 2009. 2.3

[Nai09] Mariam Nainan. Modeling interaction join point adaptations independent of pointcut models using UML stereotypes. *Proceedings of the 13th workshop on Aspect-oriented modeling*, pages 25–30, 2009. 2.4.1

[NEFE03] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12:28–63, January 2003. 2.4.1

[Obe10] Obeo. Acceleo - Transforming models into code. http://www.eclipse.org/acceleo/, 2010. 2.2.4

[Obj01] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core Specification. (formal/06-01-01), 2001. OMG Available Specification. 2.2.2, 2.2.2

[Obj09a] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification. *OMG*, 2009. 2.2.4, 2.2.4, 5.4

[Obj09b] Object Management Group (OMG). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. pages 1–738, Nov 2009. 2.2.3

[Obj10] Object Management Group (OMG). OMG Systems Modeling Language (SysML). 2010. 2.2.3

[ope10a] openArchitectureWare. openArchitectureWare (oAW). http://www.eclipse.org/workinggroups/oaw/, 2010. 2.2.3, 4.5.1

[ope10b] openArchitectureWare. Xpand, 2010. 2.2.4

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972. 1.1.1

[PVSGB08] Jens Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and Visualizing Transformation Chains. *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, pages 17–32, 2008. 2.2.5

[RB08] Stephan Roser and Bernhard Bauer. Journal on Data Semantics XI. chapter Automatic Generation and Evolution of Model Transformations Using Ontology Engineering Space, pages 32–64. Springer-Verlag, Berlin, Heidelberg, 2008. 2.3, 2.3.2

[Rea89] Chris Reade. *Elements of functional programming*. Addison-Wesley, Wokingham, UK, Jan 1989. 1.1.1, 3.3.1

[RGKK+09] Daniel Ruiz-Gonzalez, Nora Koch, Christian Kroiss, José-Raul Romero, and Antonio Vallecillo. Viewpoint synchronization of UWE models. *In Proceedings of Model-Driven Web Engineering Workshop 2009*, 2009. 8.5.1

[RHW+09] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garces, Richard F. Paige, and

Fiona A.C. Polack. An analysis of approaches to model migration. *In Proceedings of the Joint MoDSE-MCCM Workshop*, pages 6–15, 2009. 2.3.1

[RJV09] José Raùl Romero, Juan Ignacio Jaen, and Antonio Vallecillo. Realizing Correspondences in Multi-viewpoint Specifications. *Proceedings of the 2009 IEEE International Enterprise Distributed Object Computing Conference (EDOC 2009)*, pages 163–172, 2009. 5.3

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2009. (document), 2.2.2, 2.2.2, 2.3, 2.4, 6.2

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer*, 29:38–47, 1996. 1.1.1, 4.3.2, A.5

[Sch06] Douglas C Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, Feb 2006. 1.1, 2.2, 2.2.4

[SNEC06] Mehrdad Sabetzadeh, Shiva Nejati, Steve Easterbrook, and Marsha Chechik. A relationship-driven approach to view merging. *ACM SIGSOFT Software Engineering Notes*, 31:1–2, November 2006. 5.4

[SSJ02] Inderjeet Singh, Beth Stearns, and Mark Johnson. *Designing enterprise applications with the J2EE platform*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 1.1

[SSK+07] A Schauerhuber, W Schwinger, E Kapsammer, W Retschitzegger, M Wimmer, and G Kappel. A survey on aspect-oriented approaches. *Relatorio tecnico, Vienna University of Technology*, 2007. 2.4.1, 2.4.1, 3.1, 3.3

[TCJ10] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving Higher-Order Transformations Support in ATL. *Theory and Practice of Model Transformations*, 6142:215–229, 2010. 2.2.4

[THA07] Bedir Tekinerdogan, Christian Hofmann, and Mehmet Aksit. Modeling Traceability of Concerns for Synchronizing Architectural Views. *Journal of Object Technology*, 6(7):7–25, August 2007. 2.4.1

[TK05] Juha-Pekka Tolvanen and Steven Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. *Software Product Lines*, pages 198–209, 2005. 2.2.2, 2.2.3, 3.3.1

[VB05] Bert Vanhooff and Yolande Berbers. Supporting modular transformation units with precise transformation traceability metadata. *ECMDA-Traceability Workshop, SINTEF*, pages 15–27, 2005. 2.3.2

[VBJB] Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Traceability as input for model transformations. *Third ECMDA Traceability Workshop 2007*. 2.2.6, 2.2.6

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, June 2000. 2.2.3

[VdL02] Hans Vangheluwe and Juan de Lara. An introduction to multi-paradigm modelling and simulation. *In Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, pages 9–20, 2002. 2.4.2

[VVBH+06] Bert Vanhooff, Stefan Van Baelen, Aram Hovsepyan, Wouter Joosen, and Yolande Berbers. Towards a transformation chain modeling language. *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 4017:39–48, 2006. 2.2.5

[VWDD07] Eelco Visser, Jos Warmer, Arie Van Deursen, and Arie Van Deursen. Model-driven software evolution: A research agenda. *In Proceedings Int. Workshop on Model-Driven Software Evolution held with the ECSMR'07*, 2007. 2.3

[Wac07] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer Berlin / Heidelberg, 2007. 1.1.2, 2.3, 2.3.1

[Wag08] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 152–167, 2008. 6.5.1

[WVDSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software and Systems Modeling*, 9:285–309, 2010. 10.1007/s10270-009-0134-3. 2.3.2

[YCDS08] Andres Yie, Rubby Casallas, Dirk Deridder, and Ragnhild Straeten. Multi-step concern refinement. *EA '08: Proceedings of the 2008 AOSD workshop on Early aspects*, Mar 2008. 1.5, 8.3.3

[YCDW09a] Andrés Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. An approach for evolving transformation chains. *Model Driven Engineering Languages and Systems*, pages 551–555, 2009. 1.5, 8.3

[YCDW09b] Andres Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. A practical approach to multi-modeling views composition. *Electronic Communications of the EASST*, 21(Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM 2009)):1–11, Oct 2009. 1.5, 8.3

[YCDW10a] Andrés Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. Deriving correspondence relationships to guide a multi-view heterogeneous composition. *Models in Software Engineering*, 6002:225–239, 2010. 1.5, 8.3

[YCDW10b] Andrés Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. Realizing model transformation chain interoperability. *Software and Systems Modeling*, pages 1–21, 2010. 10.1007/s10270-010-0179-3. 1.5, 8.3

[YW09] Andres Yie and Dennis Wagelaar. Advanced traceability for atl. In *1st International Workshop on Model Transformation with ATL*, pages 78–87, 2009. 6, 8.3.4

Index